

## Pointers and `constexpr`

It is important to understand that when we define a pointer in a `constexpr` declaration, the `constexpr` specifier applies to the pointer, not the type to which the pointer points:

```
const int *p = nullptr;      // p is a pointer to a const int
constexpr int *q = nullptr; // q is a const pointer to int
```

Despite appearances, the types of `p` and `q` are quite different; `p` is a pointer to `const`, whereas `q` is a constant pointer. The difference is a consequence of the fact that `constexpr` imposes a top-level `const` (§ 2.4.3, p. 63) on the objects it defines.

Like any other constant pointer, a `constexpr` pointer may point to a `const` or a `nonconst` type:

```
constexpr int *np = nullptr; // np is a constant pointer to int that is null
int j = 0;
constexpr int i = 42; // type of i is const int
// i and j must be defined outside any function
constexpr const int *p = &i; // p is a constant pointer to the const int i
constexpr int *p1 = &j; // p1 is a constant pointer to the int j
```

### EXERCISES SECTION 2.4.4

**Exercise 2.32:** Is the following code legal or not? If not, how might you make it legal?

```
int null = 0, *p = null;
```

## 2.5 Dealing with Types

As our programs get more complicated, we'll see that the types we use also get more complicated. Complications in using types arise in two different ways. Some types are hard to "spell." That is, they have forms that are tedious and error-prone to write. Moreover, the form of a complicated type can obscure its purpose or meaning. The other source of complication is that sometimes it is hard to determine the exact type we need. Doing so can require us to look back into the context of the program.

### 2.5.1 Type Aliases

A **type alias** is a name that is a synonym for another type. Type aliases let us simplify complicated type definitions, making those types easier to use. Type aliases also let us emphasize the purpose for which a type is used.

We can define a type alias in one of two ways. Traditionally, we use a `typedef`:

```
typedef double wages; // wages is a synonym for double
typedef wages base, *p; // base is a synonym for double, p for double*
```

The keyword `typedef` may appear as part of the base type of a declaration (§ 2.3, p. 50). Declarations that include `typedef` define type aliases rather than variables. As in any other declaration, the declarators can include type modifiers that define compound types built from the base type of the definition.

 The new standard introduced a second way to define a type alias, via an **alias declaration**:

```
using SI = Sales_item; // SI is a synonym for Sales_item
```

An alias declaration starts with the keyword `using` followed by the alias name and an `=`. The alias declaration defines the name on the left-hand side of the `=` as an alias for the type that appears on the right-hand side.

A type alias is a type name and can appear wherever a type name can appear:

```
wages hourly, weekly; // same as double hourly, weekly;
SI item; // same as Sales_item item
```



## Pointers, `const`, and Type Aliases

Declarations that use type aliases that represent compound types and `const` can yield surprising results. For example, the following declarations use the type `pstring`, which is an alias for the type `char*`:

```
typedef char *pstring;
const pstring cstr = 0; // cstr is a constant pointer to char
const pstring *ps; // ps is a pointer to a constant pointer to char
```

The base type in these declarations is `const pstring`. As usual, a `const` that appears in the base type modifies the given type. The type of `pstring` is “pointer to `char`.” So, `const pstring` is a constant pointer to `char`—not a pointer to `const char`.

It can be tempting, albeit incorrect, to interpret a declaration that uses a type alias by conceptually replacing the alias with its corresponding type:

```
const char *cstr = 0; // wrong interpretation of const pstring cstr
```

However, this interpretation is wrong. When we use `pstring` in a declaration, the base type of the declaration is a pointer type. When we rewrite the declaration using `char*`, the base type is `char` and the `*` is part of the declarator. In this case, `const char` is the base type. This rewrite declares `cstr` as a pointer to `const char` rather than as a `const` pointer to `char`.



## 2.5.2 The `auto` Type Specifier

 It is not uncommon to want to store the value of an expression in a variable. To declare the variable, we have to know the type of that expression. When we write a program, it can be surprisingly difficult—and sometimes even impossible—to determine the type of an expression. Under the new standard, we can let the compiler figure out the type for us by using the `auto` type specifier. Unlike type specifiers, such as `double`, that name a specific type, `auto` tells the compiler to deduce

the type from the initializer. By implication, a variable that uses `auto` as its type specifier must have an initializer:

```
// the type of item is deduced from the type of the result of adding val1 and val2
auto item = val1 + val2; // item initialized to the result of val1 + val2
```

Here the compiler will deduce the type of `item` from the type returned by applying `+` to `val1` and `val2`. If `val1` and `val2` are `Sales_item` objects (§ 1.5, p. 19), `item` will have type `Sales_item`. If those variables are type `double`, then `item` has type `double`, and so on.

As with any other type specifier, we can define multiple variables using `auto`. Because a declaration can involve only a single base type, the initializers for all the variables in the declaration must have types that are consistent with each other:

```
auto i = 0, *p = &i;      // ok: i is int and p is a pointer to int
auto sz = 0, pi = 3.14;   // error: inconsistent types for sz and pi
```

## Compound Types, `const`, and `auto`

The type that the compiler infers for `auto` is not always exactly the same as the initializer's type. Instead, the compiler adjusts the type to conform to normal initialization rules.

First, as we've seen, when we use a reference, we are really using the object to which the reference refers. In particular, when we use a reference as an initializer, the initializer is the corresponding object. The compiler uses that object's type for `auto`'s type deduction:

```
int i = 0, &r = i;
auto a = r; // a is an int (r is an alias for i, which has type int)
```

Second, `auto` ordinarily ignores top-level `consts` (§ 2.4.3, p. 63). As usual in initializations, low-level `consts`, such as when an initializer is a pointer to `const`, are kept:

```
const int ci = i, &cr = ci;
auto b = ci; // b is an int (top-level const in ci is dropped)
auto c = cr; // c is an int (cr is an alias for ci whose const is top-level)
auto d = &i; // d is an int*(& of an int object is int*)
auto e = &ci; // e is const int*(& of a const object is low-level const)
```

If we want the deduced type to have a top-level `const`, we must say so explicitly:

```
const auto f = ci; // deduced type of ci is int; f has type const int
```

We can also specify that we want a reference to the `auto`-deduced type. Normal initialization rules still apply:

```
auto &g = ci; // g is a const int& that is bound to ci
auto &h = 42; // error: we can't bind a plain reference to a literal
const auto &j = 42; // ok: we can bind a const reference to a literal
```

When we ask for a reference to an auto-deduced type, top-level consts in the initializer are not ignored. As usual, consts are not top-level when we bind a reference to an initializer.

When we define several variables in the same statement, it is important to remember that a reference or pointer is part of a particular declarator and not part of the base type for the declaration. As usual, the initializers must provide consistent auto-deduced types:

```
auto k = ci, &l = i;      // k is int; l is int&
auto &m = ci, *p = &ci; // m is a const int&; p is a pointer to const int
// error: type deduced from i is int; type deduced from &ci is const int
auto &n = i, *p2 = &ci;
```

### EXERCISES SECTION 2.5.2

**Exercise 2.33:** Using the variable definitions from this section, determine what happens in each of these assignments:

```
a = 42;    b = 42;    c = 42;
d = 42;    e = 42;    g = 42;
```

**Exercise 2.34:** Write a program containing the variables and assignments from the previous exercise. Print the variables before and after the assignments to check whether your predictions in the previous exercise were correct. If not, study the examples until you can convince yourself you know what led you to the wrong conclusion.

**Exercise 2.35:** Determine the types deduced in each of the following definitions. Once you've figured out the types, write a program to see whether you were correct.

```
const int i = 42;
auto j = i; const auto &k = i; auto *p = &i;
const auto j2 = i, &k2 = i;
```



### 2.5.3 The `decltype` Type Specifier

Sometimes we want to define a variable with a type that the compiler deduces from an expression but do not want to use that expression to initialize the variable. For such cases, the new standard introduced a second type specifier, `decltype`, which returns the type of its operand. The compiler analyzes the expression to determine its type but does not evaluate the expression:

```
decltype(f()) sum = x; // sum has whatever type f returns
```

Here, the compiler does not call `f`, but it uses the type that such a call would return as the type for `sum`. That is, the compiler gives `sum` the same type as the type that would be returned if we were to call `f`.

The way `decltype` handles top-level `const` and references differs subtly from the way `auto` does. When the expression to which we apply `decltype` is a vari-

C++  
11

able, `decltype` returns the type of that variable, including top-level `const` and references:

```
const int ci = 0, &cj = ci;
decltype(ci) x = 0; // x has type const int
decltype(cj) y = x; // y has type const int& and is bound to x
decltype(cj) z;      // error: z is a reference and must be initialized
```

Because `cj` is a reference, `decltype(cj)` is a reference type. Like any other reference, `z` must be initialized.

It is worth noting that `decltype` is the *only* context in which a variable defined as a reference is not treated as a synonym for the object to which it refers.

## decltype and References



When we apply `decltype` to an expression that is not a variable, we get the type that that expression yields. As we'll see in § 4.1.1 (p. 135), some expressions will cause `decltype` to yield a reference type. Generally speaking, `decltype` returns a reference type for expressions that yield objects that can stand on the left-hand side of the assignment:

```
// decltype of an expression can be a reference type
int i = 42, *p = &i, &r = i;
decltype(r + 0) b; // ok: addition yields an int; b is an (uninitialized) int
decltype(*p) c;    // error: c is int& and must be initialized
```

Here `r` is a reference, so `decltype(r)` is a reference type. If we want the type to which `r` refers, we can use `r` in an expression, such as `r + 0`, which is an expression that yields a value that has a nonreference type.

On the other hand, the dereference operator is an example of an expression for which `decltype` returns a reference. As we've seen, when we dereference a pointer, we get the object to which the pointer points. Moreover, we can assign to that object. Thus, the type deduced by `decltype(*p)` is `int&`, not plain `int`.



Another important difference between `decltype` and `auto` is that the deduction done by `decltype` depends on the form of its given expression. What can be confusing is that enclosing the name of a variable in parentheses affects the type returned by `decltype`. When we apply `decltype` to a variable without any parentheses, we get the type of that variable. If we wrap the variable's name in one or more sets of parentheses, the compiler will evaluate the operand as an expression. A variable is an expression that can be the left-hand side of an assignment. As a result, `decltype` on such an expression yields a reference:

```
// decltype of a parenthesized variable is always a reference
decltype((i)) d; // error: d is int& and must be initialized
decltype(i) e;    // ok: e is an (uninitialized) int
```



Remember that `decltype((variable))` (note, double parentheses) is always a reference type, but `decltype(variable)` is a reference type only if `variable` is a reference.

## EXERCISES SECTION 2.5.3

**Exercise 2.36:** In the following code, determine the type of each variable and the value each variable has when the code finishes:

```
int a = 3, b = 4;
decltype(a) c = a;
decltype((b)) d = a;
++c;
++d;
```

**Exercise 2.37:** Assignment is an example of an expression that yields a reference type. The type is a reference to the type of the left-hand operand. That is, if *i* is an *int*, then the type of the expression *i* = *x* is *int&*. Using that knowledge, determine the type and value of each variable in this code:

```
int a = 3, b = 4;
decltype(a) c = a;
decltype(a = b) d = a;
```

**Exercise 2.38:** Describe the differences in type deduction between `decltype` and `auto`. Give an example of an expression where `auto` and `decltype` will deduce the same type and an example where they will deduce differing types.



## 2.6 Defining Our Own Data Structures

At the most basic level, a data structure is a way to group together related data elements and a strategy for using those data. As one example, our `Sales_item` class groups an ISBN, a count of how many copies of that book had been sold, and the revenue associated with those sales. It also provides a set of operations such as the `isbn` function and the `>>`, `<<`, `+`, and `+=` operators.

In C++ we define our own data types by defining a class. The library types `string`, `istream`, and `ostream` are all defined as classes, as is the `Sales_item` type we used in Chapter 1. C++ support for classes is extensive—in fact, Parts III and IV are largely devoted to describing class-related features. Even though the `Sales_item` class is pretty simple, we won’t be able to fully define that class until we learn how to write our own operators in Chapter 14.



### 2.6.1 Defining the `Sales_data` Type

Although we can’t yet write our `Sales_item` class, we can write a more concrete class that groups the same data elements. Our strategy for using this class is that users will be able to access the data elements directly and must implement needed operations for themselves.

Because our data structure does not support any operations, we’ll name our version `Sales_data` to distinguish it from `Sales_item`. We’ll define our class as follows:

```
struct Sales_data {  
    std::string bookNo;  
    unsigned units_sold = 0;  
    double revenue = 0.0;  
};
```

Our class begins with the keyword **struct**, followed by the name of the class and a (possibly empty) class body. The class body is surrounded by curly braces and forms a new scope (§ 2.2.4, p. 48). The names defined inside the class must be unique within the class but can reuse names defined outside the class.

The close curly that ends the class body must be followed by a semicolon. The semicolon is needed because we can define variables after the class body:

```
struct Sales_data { /* ... */ } accum, trans, *salesptr;  
// equivalent, but better way to define these objects  
struct Sales_data { /* ... */ };  
Sales_data accum, trans, *salesptr;
```

The semicolon marks the end of the (usually empty) list of declarators. Ordinarily, it is a bad idea to define an object as part of a class definition. Doing so obscures the code by combining the definitions of two different entities—the class and a variable—in a single statement.



It is a common mistake among new programmers to forget the semicolon at the end of a class definition.

## Class Data Members

The class body defines the **members** of the class. Our class has only **data members**. The data members of a class define the contents of the objects of that class type. Each object has its own copy of the class data members. Modifying the data members of one object does not change the data in any other `Sales_data` object.

We define data members the same way that we define normal variables: We specify a base type followed by a list of one or more declarators. Our class has three data members: a member of type `string` named `bookNo`, an `unsigned` member named `units_sold`, and a member of type `double` named `revenue`. Each `Sales_data` object will have these three data members.

Under the new standard, we can supply an **in-class initializer** for a data member. When we create objects, the in-class initializers will be used to initialize the data members. Members without an initializer are default initialized (§ 2.2.1, p. 43). Thus, when we define `Sales_data` objects, `units_sold` and `revenue` will be initialized to 0, and `bookNo` will be initialized to the empty string.

C++  
11

In-class initializers are restricted as to the form (§ 2.2.1, p. 43) we can use: They must either be enclosed inside curly braces or follow an `=` sign. We may not specify an in-class initializer inside parentheses.

In § 7.2 (p. 268), we'll see that C++ has a second keyword, `class`, that can be used to define our own data structures. We'll explain in that section why we use `struct` here. Until we cover additional class-related features in Chapter 7, you should use `struct` to define your own data structures.

## EXERCISES SECTION 2.6.1

**Exercise 2.39:** Compile the following program to see what happens when you forget the semicolon after a class definition. Remember the message for future reference.

```
struct Foo { /* empty */ } // Note: no semicolon
int main()
{
    return 0;
}
```

**Exercise 2.40:** Write your own version of the `Sales_data` class.



## 2.6.2 Using the `Sales_data` Class

Unlike the `Sales_item` class, our `Sales_data` class does not provide any operations. Users of `Sales_data` have to write whatever operations they need. As an example, we'll write a version of the program from § 1.5.2 (p. 23) that printed the sum of two transactions. The input to our program will be transactions such as

```
0-201-78345-X 3 20.00
0-201-78345-X 2 25.00
```

Each transaction holds an ISBN, the count of how many books were sold, and the price at which each book was sold.

### Adding Two `Sales_data` Objects

Because `Sales_data` provides no operations, we will have to write our own code to do the input, output, and addition operations. We'll assume that our `Sales_data` class is defined inside `Sales_data.h`. We'll see how to define this header in § 2.6.3 (p. 76).

Because this program will be longer than any we've written so far, we'll explain it in separate parts. Overall, our program will have the following structure:

```
#include <iostream>
#include <string>
#include "Sales_data.h"

int main()
{
    Sales_data data1, data2;
    // code to read into data1 and data2
    // code to check whether data1 and data2 have the same ISBN
    //      and if so print the sum of data1 and data2
}
```

As in our original program, we begin by including the headers we'll need and define variables to hold the input. Note that unlike the `Sales_item` version, our new program includes the `string` header. We need that header because our code will have to manage the `bookNo` member, which has type `string`.

## Reading Data into a `Sales_data` Object

Although we won't describe the library `string` type in detail until Chapters 3 and 10, we need to know only a little bit about `strings` in order to define and use our `ISBN` member. The `string` type holds a sequence of characters. Its operations include the `>>`, `<<`, and `==` operators to read, write, and compare `strings`, respectively. With this knowledge we can write the code to read the first transaction:

```
double price = 0; // price per book, used to calculate total revenue
// read the first transaction: ISBN, number of books sold, price per book
std::cin >> data1.bookNo >> data1.units_sold >> price;
// calculate total revenue from price and units_sold
data1.revenue = data1.units_sold * price;
```

Our transactions contain the price at which each book was sold but our data structure stores the total revenue. We'll read the transaction data into a `double` named `price`, from which we'll calculate the `revenue` member. The `input` statement

```
std::cin >> data1.bookNo >> data1.units_sold >> price;
```

uses the dot operator (§ 1.5.2, p. 23) to read into the `bookNo` and `units_sold` members of the object named `data1`.

The last statement assigns the product of `data1.units_sold` and `price` into the `revenue` member of `data1`.

Our program will next repeat the same code to read data into `data2`:

```
// read the second transaction
std::cin >> data2.bookNo >> data2.units_sold >> price;
data2.revenue = data2.units_sold * price;
```

## Printing the Sum of Two `Sales_data` Objects

Our other task is to check that the transactions are for the same `ISBN`. If so, we'll print their sum, otherwise, we'll print an error message:

```
if (data1.bookNo == data2.bookNo) {
    unsigned totalCnt = data1.units_sold + data2.units_sold;
    double totalRevenue = data1.revenue + data2.revenue;
    // print: ISBN, total sold, total revenue, average price per book
    std::cout << data1.bookNo << " " << totalCnt
        << " " << totalRevenue << " ";
    if (totalCnt != 0)
        std::cout << totalRevenue/totalCnt << std::endl;
    else
        std::cout << "(no sales)" << std::endl;
    return 0; // indicate success
} else { // transactions weren't for the same ISBN
    std::cerr << "Data must refer to the same ISBN"
        << std::endl;
    return -1; // indicate failure
}
```

In the first `if` we compare the `bookNo` members of `data1` and `data2`. If those members are the same ISBN, we execute the code inside the curly braces. That code adds the components of our two variables. Because we'll need to print the average price, we start by computing the total of `units_sold` and `revenue` and store those in `totalCnt` and `totalRevenue`, respectively. We print those values. Next we check that there were books sold and, if so, print the computed average price per book. If there were no sales, we print a message noting that fact.

### EXERCISES SECTION 2.6.2

**Exercise 2.41:** Use your `Sales_data` class to rewrite the exercises in § 1.5.1 (p. 22), § 1.5.2 (p. 24), and § 1.6 (p. 25). For now, you should define your `Sales_data` class in the same file as your `main` function.



### 2.6.3 Writing Our Own Header Files

Although as we'll see in § 19.7 (p. 852), we can define a class inside a function, such classes have limited functionality. As a result, classes ordinarily are not defined inside functions. When we define a class outside of a function, there may be only one definition of that class in any given source file. In addition, if we use a class in several different files, the class' definition must be the same in each file.

In order to ensure that the class definition is the same in each file, classes are usually defined in header files. Typically, classes are stored in headers whose name derives from the name of the class. For example, the `string` library type is defined in the `string` header. Similarly, as we've already seen, we will define our `Sales_data` class in a header file named `Sales_data.h`.

Headers (usually) contain entities (such as class definitions and `const` and `constexpr` variables (§ 2.4, p. 60)) that can be defined only once in any given file. However, headers often need to use facilities from other headers. For example, because our `Sales_data` class has a `string` member, `Sales_data.h` must `#include` the `string` header. As we've seen, programs that use `Sales_data` also need to include the `string` header in order to use the `bookNo` member. As a result, programs that use `Sales_data` will include the `string` header twice: once directly and once as a side effect of including `Sales_data.h`. Because a header might be included more than once, we need to write our headers in a way that is safe even if the header is included multiple times.



Whenever a header is updated, the source files that use that header must be recompiled to get the new or changed declarations.

## A Brief Introduction to the Preprocessor

The most common technique for making it safe to include a header multiple times relies on the **preprocessor**. The preprocessor—which C++ inherits from C—is a

program that runs before the compiler and changes the source text of our programs. Our programs already rely on one preprocessor facility, `#include`. When the preprocessor sees a `#include`, it replaces the `#include` with the contents of the specified header.

C++ programs also use the preprocessor to define **header guards**. Header guards rely on preprocessor variables (§ 2.3.2, p. 53). Preprocessor variables have one of two possible states: defined or not defined. The `#define` directive takes a name and defines that name as a preprocessor variable. There are two other directives that test whether a given preprocessor variable has or has not been defined: `#ifdef` is true if the variable has been defined, and `#ifndef` is true if the variable has *not* been defined. If the test is true, then everything following the `#ifdef` or `#ifndef` is processed up to the matching `#endif`.

We can use these facilities to guard against multiple inclusion as follows:

```
#ifndef SALES_DATA_H
#define SALES_DATA_H
#include <string>
struct Sales_data {
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
};
#endif
```

The first time `Sales_data.h` is included, the `#ifndef` test will succeed. The preprocessor will process the lines following `#ifndef` up to the `#endif`. As a result, the preprocessor variable `SALES_DATA_H` will be defined and the contents of `Sales_data.h` will be copied into our program. If we include `Sales_data.h` later on in the same file, the `#ifndef` directive will be false. The lines between it and the `#endif` directive will be ignored.



**WARNING**

Preprocessor variable names do not respect C++ scoping rules.



Headers should have guards, even if they aren't (yet) included by another header. Header guards are trivial to write, and by habitually defining them you don't need to decide whether they are needed.

## EXERCISES SECTION 2.6.3

**Exercise 2.42:** Write your own version of the `Sales_data.h` header and use it to rewrite the exercise from § 2.6.2 (p. 76).



## 3.4 Introducing Iterators

Although we can use subscripts to access the characters of a `string` or the elements in a `vector`, there is a more general mechanism—known as **iterators**—that we can use for the same purpose. As we'll see in Part II, in addition to `vector`, the library defines several other kinds of containers. All of the library containers have iterators, but only a few of them support the subscript operator. Technically speaking, a `string` is not a container type, but `string` supports many of the container operations. As we've seen `string`, like `vector` has a subscript operator. Like `vectors`, `strings` also have iterators.

Like pointers (§ 2.3.2, p. 52), iterators give us indirect access to an object. In the case of an iterator, that object is an element in a container or a character in a `string`. We can use an iterator to fetch an element and iterators have operations to move from one element to another. As with pointers, an iterator may be valid or invalid. A valid iterator either denotes an element or denotes a position one past the last element in a container. All other iterator values are invalid.



### 3.4.1 Using Iterators

Unlike pointers, we do not use the address-of operator to obtain an iterator. Instead, types that have iterators have members that return iterators. In particular, these types have members named `begin` and `end`. The `begin` member returns an iterator that denotes the first element (or first character), if there is one:

```
// the compiler determines the type of b and e; see § 2.5.2 (p. 68)
// b denotes the first element and e denotes one past the last element in v
auto b = v.begin(), e = v.end(); // b and e have the same type
```

The iterator returned by `end` is an iterator positioned “one past the end” of the associated container (or `string`). This iterator denotes a nonexistent element “off the end” of the container. It is used as a marker indicating when we have processed all the elements. The iterator returned by `end` is often referred to as the **off-the-end iterator** or abbreviated as “the end iterator.” If the container is empty, `begin` returns the same iterator as the one returned by `end`.



If the container is empty, the iterators returned by `begin` and `end` are equal—they are both off-the-end iterators.

In general, we do not know (or care about) the precise type that an iterator has. In this example, we used `auto` to define `b` and `e` (§ 2.5.2, p. 68). As a result, these variables have whatever type is returned by the `begin` and `end` members, respectively. We'll have more to say about those types on page 108.

### Iterator Operations

Iterators support only a few operations, which are listed in Table 3.6. We can compare two valid iterators using `==` or `!=`. Iterators are equal if they denote the same element or if they are both off-the-end iterators for the same container. Otherwise, they are unequal.

As with pointers, we can dereference an iterator to obtain the element denoted by an iterator. Also, like pointers, we may dereference only a valid iterator that denotes an element (§ 2.3.2, p. 53). Dereferencing an invalid iterator or an off-the-end iterator has undefined behavior.

As an example, we'll rewrite the program from § 3.2.3 (p. 94) that capitalized the first character of a `string` using an iterator instead of a subscript:

```
string s("some string");
if (s.begin() != s.end()) { // make sure s is not empty
    auto it = s.begin(); // it denotes the first character in s
    *it = toupper(*it); // make that character uppercase
}
```

As in our original program, we first check that `s` isn't empty. In this case, we do so by comparing the iterators returned by `begin` and `end`. Those iterators are equal if the `string` is empty. If they are unequal, there is at least one character in `s`.

Inside the `if` body, we obtain an iterator to the first character by assigning the iterator returned by `begin` to `it`. We dereference that iterator to pass that character to `toupper`. We also dereference `it` on the left-hand side of the assignment in order to assign the character returned from `toupper` to the first character in `s`. As in our original program, the output of this loop will be:

`Some string`

**Table 3.6: Standard Container Iterator Operations**

<code>*iter</code>	Returns a reference to the element denoted by the iterator <code>iter</code> .
<code>iter-&gt;mem</code>	Dereferences <code>iter</code> and fetches the member named <code>mem</code> from the underlying element. Equivalent to <code>(*iter).mem</code> .
<code>++iter</code>	Increments <code>iter</code> to refer to the next element in the container.
<code>--iter</code>	Decrements <code>iter</code> to refer to the previous element in the container.
<code>iter1 == iter2</code>	Compares two iterators for equality (inequality). Two iterators are equal if they denote the same element or if they are the off-the-end iterator for the same container.
<code>iter1 != iter2</code>	

## Moving Iterators from One Element to Another

Iterators use the increment (`++`) operator (§ 1.4.1, p. 12) to move from one element to the next. Incrementing an iterator is a logically similar operation to incrementing an integer. In the case of integers, the effect is to “add 1” to the integer’s value. In the case of iterators, the effect is to “advance the iterator by one position.”



Because the iterator returned from `end` does not denote an element, it may not be incremented or dereferenced.

Using the increment operator, we can rewrite our program that changed the case of the first word in a `string` to use iterators instead:

```
// process characters in s until we run out of characters or we hit a whitespace
for (auto it = s.begin(); it != s.end() && !isspace(*it); ++it)
    *it = toupper(*it); // capitalize the current character
```

This loop, like the one in § 3.2.3 (p. 94), iterates through the characters in `s`, stopping when we encounter a whitespace character. However, this loop accesses these characters using an iterator, not a subscript.

The loop starts by initializing `it` from `s.begin`, meaning that `it` denotes the first character (if any) in `s`. The condition checks whether `it` has reached the end of `s`. If not, the condition next dereferences `it` to pass the current character to `isspace` to see whether we're done. At the end of each iteration, we execute `++it` to advance the iterator to access the next character in `s`.

The body of this loop, is the same as the last statement in the previous `if`. We dereference `it` to pass the current character to `toupper` and assign the resulting uppercase letter back into the character denoted by `it`.

### KEY CONCEPT: GENERIC PROGRAMMING

Programmers coming to C++ from C or Java might be surprised that we used `!=` rather than `<` in our `for` loops such as the one above and in the one on page 94. C++ programmers use `!=` as a matter of habit. They do so for the same reason that they use iterators rather than subscripts: This coding style applies equally well to various kinds of containers provided by the library.

As we've seen, only a few library types, `vector` and `string` being among them, have the subscript operator. Similarly, all of the library containers have iterators that define the `==` and `!=` operators. Most of those iterators do not have the `<` operator. By routinely using iterators and `!=`, we don't have to worry about the precise type of container we're processing.

## Iterator Types

Just as we do not know the precise type of a `vector`'s or `string`'s `size_type` member (§ 3.2.2, p. 88), so too, we generally do not know—and do not need to know—the precise type of an iterator. Instead, as with `size_type`, the library types that have iterators define types named `iterator` and `const_iterator` that represent actual iterator types:

```
vector<int>::iterator it; // it can read and write vector<int> elements
string::iterator it2;      // it2 can read and write characters in a string
vector<int>::const_iterator it3; // it3 can read but not write elements
string::const_iterator it4; // it4 can read but not write characters
```

A `const_iterator` behaves like a `const` pointer (§ 2.4.2, p. 62). Like a `const` pointer, a `const_iterator` may read but not write the element it denotes; an object of type `iterator` can both read and write. If a `vector` or `string` is `const`, we may use only its `const_iterator` type. With a nonconst `vector` or `string`, we can use either `iterator` or `const_iterator`.

### TERMINOLOGY: ITERATORS AND ITERATOR TYPES

The term **iterator** is used to refer to three different entities. We might mean the *concept* of an iterator, or we might refer to the **iterator type** defined by a container, or we might refer to an *object* as an iterator.

What's important to understand is that there is a collection of types that are related conceptually. A type is an iterator if it supports a common set of actions. Those actions let us access an element in a container and let us move from one element to another.

Each container class defines a type named **iterator**; that **iterator** type supports the actions of an (conceptual) iterator.

## The `begin` and `end` Operations

The type returned by `begin` and `end` depends on whether the object on which they operator is `const`. If the object is `const`, then `begin` and `end` return a `const_iterator`; if the object is not `const`, they return `iterator`:

```
vector<int> v;
const vector<int> cv;
auto it1 = v.begin(); // it1 has type vector<int>::iterator
auto it2 = cv.begin(); // it2 has type vector<int>::const_iterator
```

Often this default behavior is not what we want. For reasons we'll explain in § 6.2.3 (p. 213), it is usually best to use a `const` type (such as `const_iterator`) when we need to read but do not need to write to an object. To let us ask specifically for the `const_iterator` type, the new standard introduced two new functions named `cbegin` and `cend`:

```
auto it3 = v.cbegin(); // it3 has type vector<int>::const_iterator
```

As do the `begin` and `end` members, these members return iterators to the first and one past the last element in the container. However, regardless of whether the `vector` (or `string`) is `const`, they return a `const_iterator`.

C++  
11

## Combining Dereference and Member Access

When we dereference an iterator, we get the object that the iterator denotes. If that object has a class type, we may want to access a member of that object. For example, we might have a `vector` of `strings` and we might need to know whether a given element is empty. Assuming `it` is an iterator into this `vector`, we can check whether the `string` that `it` denotes is empty as follows:

```
(*it).empty()
```

For reasons we'll cover in § 4.1.2 (p. 136), the parentheses in `(*it).empty()` are necessary. The parentheses say to apply the dereference operator to `it` and to apply the dot operator (§ 1.5.2, p. 23) to the result of dereferencing `it`. Without parentheses, the dot operator would apply to `it`, not to the resulting object:

```
(*it).empty() // dereferences it and calls the member empty on the resulting object
*it.empty()    // error: attempts to fetch the member named empty from it
                //       but it is an iterator and has no member named empty
```

The second expression is interpreted as a request to fetch the `empty` member from the object named `it`. However, `it` is an iterator and has no member named `empty`. Hence, the second expression is in error.

To simplify expressions such as this one, the language defines the arrow operator (the `->` operator). The arrow operator combines dereference and member access into a single operation. That is, `it->mem` is a synonym for `(*it).mem`.

For example, assume we have a `vector<string>` named `text` that holds the data from a text file. Each element in the vector is either a sentence or an empty string representing a paragraph break. If we want to print the contents of the first paragraph from `text`, we'd write a loop that iterates through `text` until we encounter an element that is empty:

```
// print each line in text up to the first blank line
for (auto it = text.cbegin();
     it != text.cend() && !it->empty(); ++it)
    cout << *it << endl;
```

We start by initializing `it` to denote the first element in `text`. The loop continues until either we process every element in `text` or we find an element that is empty. So long as there are elements and we haven't seen an empty element, we print the current element. It is worth noting that because the loop reads but does not write to the elements in `text`, we use `cbegin` and `cend` to control the iteration.

## Some vector Operations Invalidate Iterators

In § 3.3.2 (p. 101) we noted that there are implications of the fact that vectors can grow dynamically. We also noted that one such implication is that we cannot add elements to a vector inside a range for loop. Another implication is that any operation, such as `push_back`, that changes the size of a vector potentially invalidates all iterators into that vector. We'll explore how iterators become invalid in more detail in § 9.3.6 (p. 353).



For now, it is important to realize that loops that use iterators should not add elements to the container to which the iterators refer.

### EXERCISES SECTION 3.4.1

**Exercise 3.21:** Redo the first exercise from § 3.3.3 (p. 105) using iterators.

**Exercise 3.22:** Revise the loop that printed the first paragraph in `text` to instead change the elements in `text` that correspond to the first paragraph to all uppercase. After you've updated `text`, print its contents.

**Exercise 3.23:** Write a program to create a vector with ten `int` elements. Using an iterator, assign each element a value that is twice its current value. Test your program by printing the vector.

### 3.4.2 Iterator Arithmetic



Incrementing an iterator moves the iterator one element at a time. All the library containers have iterators that support increment. Similarly, we can use `==` and `!=` to compare two valid iterators (§ 3.4, p. 106) into any of the library container types.

Iterators for `string` and `vector` support additional operations that can move an iterator multiple elements at a time. They also support all the relational operators. These operations, which are often referred to as **iterator arithmetic**, are described in Table 3.7.

**Table 3.7: Operations Supported by `vector` and `string` Iterators**

<code>iter + n</code>	Adding (subtracting) an integral value <code>n</code> to (from) an iterator yields an iterator that many elements forward (backward) within the container. The resulting iterator must denote elements in, or one past the end of, the same container.
<code>iter - n</code>	
<code>iter1 += n</code>	Compound-assignment for iterator addition and subtraction. Assigns to <code>iter1</code> the value of adding <code>n</code> to, or subtracting <code>n</code> from, <code>iter1</code> .
<code>iter1 -= n</code>	
<code>iter1 - iter2</code>	Subtracting two iterators yields the number that when added to the right-hand iterator yields the left-hand iterator. The iterators must denote elements in, or one past the end of, the same container.
<code>&gt;, &gt;=, &lt;, &lt;=</code>	Relational operators on iterators. One iterator is less than another if it refers to an element that appears in the container before the one referred to by the other iterator. The iterators must denote elements in, or one past the end of, the same container.

### Arithmetic Operations on Iterators

We can add (or subtract) an integral value and an iterator. Doing so returns an iterator positioned forward (or backward) that many elements. When we add or subtract an integral value and an iterator, the result must denote an element in the same `vector` (or `string`) or denote one past the end of the associated `vector` (or `string`). As an example, we can compute an iterator to the element nearest the middle of a `vector`:

```
// compute an iterator to the element closest to the midpoint of vi
auto mid = vi.begin() + vi.size() / 2;
```

If `vi` has 20 elements, then `vi.size() / 2` is 10. In this case, we'd set `mid` equal to `vi.begin() + 10`. Remembering that subscripts start at 0, this element is the same as `vi[10]`, the element ten past the first.

In addition to comparing two iterators for equality, we can compare `vector` and `string` iterators using the relational operators (`<`, `<=`, `>`, `>=`). The iterators must be valid and must denote elements in (or one past the end of) the same `vector` or `string`. For example, assuming `it` is an iterator into the same `vector` as `mid`, we can check whether `it` denotes an element before or after `mid` as follows:

```
if (it < mid)
    // process elements in the first half of vi
```

We can also subtract two iterators so long as they refer to elements in, or one off the end of, the same vector or string. The result is the distance between the iterators. By distance we mean the amount by which we'd have to change one iterator to get the other. The result type is a signed integral type named `difference_type`. Both `vector` and `string` define `difference_type`. This type is signed, because subtraction might have a negative result.

## Using Iterator Arithmetic

A classic algorithm that uses iterator arithmetic is binary search. A binary search looks for a particular value in a sorted sequence. It operates by looking at the element closest to the middle of the sequence. If that element is the one we want, we're done. Otherwise, if that element is smaller than the one we want, we continue our search by looking only at elements after the rejected one. If the middle element is larger than the one we want, we continue by looking only in the first half. We compute a new middle element in the reduced range and continue looking until we either find the element or run out of elements.

We can do a binary search using iterators as follows:

```
// text must be sorted
// beg and end will denote the range we're searching
auto beg = text.begin(), end = text.end();
auto mid = text.begin() + (end - beg)/2; // original midpoint
// while there are still elements to look at and we haven't yet found sought
while (mid != end && *mid != sought) {
    if (sought < *mid)      // is the element we want in the first half?
        end = mid;          // if so, adjust the range to ignore the second half
    else                      // the element we want is in the second half
        beg = mid + 1;       // start looking with the element just after mid
    mid = beg + (end - beg)/2; // new midpoint
}
```

We start by defining three iterators: `beg` will be the first element in the range, `end` one past the last element, and `mid` the element closest to the middle. We initialize these iterators to denote the entire range in a `vector<string>` named `text`.

Our loop first checks that the range is not empty. If `mid` is equal to the current value of `end`, then we've run out of elements to search. In this case, the condition fails and we exit the `while`. Otherwise, `mid` refers to an element and we check whether `mid` denotes the one we want. If so, we're done and we exit the loop.

If we still have elements to process, the code inside the `while` adjusts the range by moving `end` or `beg`. If the element denoted by `mid` is greater than `sought`, we know that if `sought` is in `text`, it will appear before the element denoted by `mid`. Therefore, we can ignore elements after `mid`, which we do by assigning `mid` to `end`. If `*mid` is smaller than `sought`, the element must be in the range of elements after the one denoted by `mid`. In this case, we adjust the range by making `beg` denote the element just after `mid`. We already know that `mid` is not the one we want, so we can eliminate it from the range.

At the end of the `while`, `mid` will be equal to `end` or it will denote the element for which we are looking. If `mid` equals `end`, then the element was not in `text`.

*The fundamental ideas* behind **classes** are **data abstraction** and **encapsulation**. Data abstraction is a programming (and design) technique that relies on the separation of **interface** and **implementation**. The interface of a class consists of the operations that users of the class can execute. The implementation includes the class' data members, the bodies of the functions that constitute the interface, and any functions needed to define the class that are not intended for general use.

Encapsulation enforces the separation of a class' interface and implementation. A class that is encapsulated hides its implementation—users of the class can use the interface but have no access to the implementation.

A class that uses data abstraction and encapsulation defines an **abstract data type**. In an abstract data type, the class designer worries about how the class is implemented. Programmers who use the class need not know how the type works. They can instead think *abstractly* about what the type does.

## 7.1 Defining Abstract Data Types

The `Sales_item` class that we used in Chapter 1 is an abstract data type. We use a `Sales_item` object by using its interface (i.e., the operations described in § 1.5.1 (p. 20)). We have no access to the data members stored in a `Sales_item` object. Indeed, we don't even know what data members that class has.

Our `Sales_data` class (§ 2.6.1, p. 72) is not an abstract data type. It lets users of the class access its data members and forces users to write their own operations. To make `Sales_data` an abstract type, we need to define operations for users of `Sales_data` to use. Once `Sales_data` defines its own operations, we can encapsulate (that is, hide) its data members.



### 7.1.1 Designing the `Sales_data` Class

Ultimately, we want `Sales_data` to support the same set of operations as the `Sales_item` class. The `Sales_item` class had one **member function** (§ 1.5.2, p. 23), named `isbn`, and supported the `+`, `=`, `+=`, `<<`, and `>>` operators.

We'll learn how to define our own operators in Chapter 14. For now, we'll define ordinary (named) functions for these operations. For reasons that we will explain in § 14.1 (p. 555), the functions that do addition and IO will not be members of `Sales_data`. Instead, we'll define those functions as ordinary functions. The function that handles compound assignment will be a member, and for reasons we'll explain in § 7.1.5 (p. 267), our class doesn't need to define assignment.

Thus, the interface to `Sales_data` consists of the following operations:

- An `isbn` member function to return the object's ISBN
- A `combine` member function to add one `Sales_data` object into another
- A function named `add` to add two `Sales_data` objects
- A `read` function to read data from an `istream` into a `Sales_data` object
- A `print` function to print the value of a `Sales_data` object on an `ostream`

### KEY CONCEPT: DIFFERENT KINDS OF PROGRAMMING ROLES

Programmers tend to think about the people who will run their applications as *users*. Similarly a class designer designs and implements a class for *users* of that class. In this case, the user is a programmer, not the ultimate user of the application.

When we refer to a *user*, the context makes it clear which kind of user is meant. If we speak of *user code* or the *user* of the `Sales_data` class, we mean a programmer who is using a class. If we speak of the *user* of the bookstore application, we mean the manager of the store who is running the application.



C++ programmers tend to speak of *users* interchangeably as users of the application or users of a class.

In simple applications, the user of a class and the designer of the class might be one and the same person. Even in such cases, it is useful to keep the roles distinct. When we design the interface of a class, we should think about how easy it will be to use the class. When we use the class, we shouldn't think about how the class works.

Authors of successful applications do a good job of understanding and implementing the needs of the application's users. Similarly, good class designers pay close attention to the needs of the programmers who will use the class. A well-designed class has an interface that is intuitive and easy to use and has an implementation that is efficient enough for its intended use.

## Using the Revised `Sales_data` Class

Before we think about how to implement our class, let's look at how we can use our interface functions. As one example, we can use these functions to write a version of the bookstore program from § 1.6 (p. 24) that works with `Sales_data` objects rather than `Sales_items`:

```
Sales_data total;           // variable to hold the running sum
if (read(cin, total)) {    // read the first transaction
    Sales_data trans;      // variable to hold data for the next transaction
    while(read(cin, trans)) { // read the remaining transactions
        if (total.isbn() == trans.isbn()) // check the isbn
            total.combine(trans); // update the running total
        else {
            print(cout, total) << endl; // print the results
            total = trans;             // process the next book
        }
    }
    print(cout, total);          // print the last transaction
} else {
    cerr << "No data?!" << endl; // there was no input
}
```

We start by defining a `Sales_data` object to hold the running total. Inside the `if` condition, we call `read` to read the first transaction into `total`. This condition works like other loops we've written that used the `>>` operator. Like the `>>` operator, our `read` function will return its stream parameter, which the condition

checks (§ 4.11.2, p. 162). If the `read` fails, we fall through to the `else` to print an error message.

If there are data to read, we define `trans`, which we'll use to hold each transaction. The condition in the `while` also checks the stream returned by `read`. So long as the input operations in `read` succeed, the condition succeeds and we have another transaction to process.

Inside the `while`, we call the `isbn` members of `total` and `trans` to fetch their respective ISBNs. If `total` and `trans` refer to the same book, we call `combine` to add the components of `trans` into the running total in `total`. If `trans` represents a new book, we call `print` to print the total for the previous book. Because `print` returns a reference to its `stream` parameter, we can use the result of `print` as the left-hand operand of the `<<`. We do so to print a newline following the output generated by `print`. We next assign `trans` to `total`, thus setting up to process the records for the next book in the file.

After we have exhausted the input, we have to remember to print the data for the last transaction, which we do in the call to `print` following the `while` loop.

## EXERCISES SECTION 7.1.1

**Exercise 7.1:** Write a version of the transaction-processing program from § 1.6 (p. 24) using the `Sales_data` class you defined for the exercises in § 2.6.1 (p. 72).



### 7.1.2 Defining the Revised `Sales_data` Class

Our revised class will have the same data members as the version we defined in § 2.6.1 (p. 72): `bookNo`, a `string` representing the ISBN; `units_sold`, an `unsigned` that says how many copies of the book were sold; and `revenue`, a `double` representing the total revenue for those sales.

As we've seen, our class will also have two member functions, `combine` and `isbn`. In addition, we'll give `Sales_data` another member function to return the average price at which the books were sold. This function, which we'll name `avg_price`, isn't intended for general use. It will be part of the implementation, not part of the interface.

We define (§ 6.1, p. 202) and declare (§ 6.1.2, p. 206) member functions similarly to ordinary functions. Member functions *must* be declared inside the class. Member functions *may* be defined inside the class itself or outside the class body. Non-member functions that are part of the interface, such as `add`, `read`, and `print`, are declared and defined outside the class.

With this knowledge, we're ready to write our revised version of `Sales_data`:

```
struct Sales_data {
    // new members: operations on Sales_data objects
    std::string isbn() const { return bookNo; }
    Sales_data& combine(const Sales_data&);
    double avg_price() const;
```

```
// data members are unchanged from § 2.6.1 (p. 72)
std::string bookNo;
unsigned units_sold = 0;
double revenue = 0.0;
};

// nonmember Sales_data interface functions
Sales_data add(const Sales_data&, const Sales_data&);
std::ostream &print(std::ostream&, const Sales_data&);
std::istream &read(std::istream&, Sales_data&);
```



Functions defined in the class are implicitly `inline` (§ 6.5.2, p. 238).

## Defining Member Functions

Although every member must be declared inside its class, we can define a member function's body either inside or outside of the class body. In `Sales_data`, `isbn` is defined inside the class; `combine` and `avg_price` will be defined elsewhere.

We'll start by explaining the `isbn` function, which returns a `string` and has an empty parameter list:

```
std::string isbn() const { return bookNo; }
```

As with any function, the body of a member function is a block. In this case, the block contains a single `return` statement that returns the `bookNo` data member of a `Sales_data` object. The interesting thing about this function is how it gets the object from which to fetch the `bookNo` member.

## Introducing `this`



Let's look again at a call to the `isbn` member function:

```
total.isbn()
```

Here we use the dot operator (§ 4.6, p. 150) to fetch the `isbn` member of the object named `total`, which we then call.

With one exception that we'll cover in § 7.6 (p. 300), when we call a member function we do so on behalf of an object. When `isbn` refers to members of `Sales_data` (e.g., `bookNo`), it is referring implicitly to the members of the object on which the function was called. In this call, when `isbn` returns `bookNo`, it is implicitly returning `total.bookNo`.

Member functions access the object on which they were called through an extra, implicit parameter named `this`. When we call a member function, `this` is initialized with the address of the object on which the function was invoked. For example, when we call

```
total.isbn()
```

the compiler passes the address of `total` to the implicit `this` parameter in `isbn`. It is as if the compiler rewrites this call as

---

```
// pseudo-code illustration of how a call to a member function is translated
Sales_data::isbn(&total)
```

which calls the `isbn` member of `Sales_data` passing the address of `total`.

Inside a member function, we can refer directly to the members of the object on which the function was called. We do not have to use a member access operator to use the members of the object to which `this` points. Any direct use of a member of the class is assumed to be an implicit reference through `this`. That is, when `isbn` uses `bookNo`, it is implicitly using the member to which `this` points. It is as if we had written `this->bookNo`.

The `this` parameter is defined for us implicitly. Indeed, it is illegal for us to define a parameter or variable named `this`. Inside the body of a member function, we can use `this`. It would be legal, although unnecessary, to define `isbn` as

```
std::string isbn() const { return this->bookNo; }
```

Because `this` is intended to always refer to “`this`” object, `this` is a `const` pointer (§ 2.4.2, p. 62). We cannot change the address that `this` holds.

## Introducing `const` Member Functions

The other important part about the `isbn` function is the keyword `const` that follows the parameter list. The purpose of that `const` is to modify the type of the implicit `this` pointer.

By default, the type of `this` is a `const` pointer to the nonconst version of the class type. For example, by default, the type of `this` in a `Sales_data` member function is `Sales_data *const`. Although `this` is implicit, it follows the normal initialization rules, which means that (by default) we cannot bind `this` to a `const` object (§ 2.4.2, p. 62). This fact, in turn, means that we cannot call an ordinary member function on a `const` object.

If `isbn` were an ordinary function and if `this` were an ordinary pointer parameter, we would declare `this` as `const Sales_data *const`. After all, the body of `isbn` doesn’t change the object to which `this` points, so our function would be more flexible if `this` were a pointer to `const` (§ 6.2.3, p. 213).

However, `this` is implicit and does not appear in the parameter list. There is no place to indicate that `this` should be a pointer to `const`. The language resolves this problem by letting us put `const` after the parameter list of a member function. A `const` following the parameter list indicates that `this` is a pointer to `const`. Member functions that use `const` in this way are **`const` member functions**.

We can think of the body of `isbn` as if it were written as

```
// pseudo-code illustration of how the implicit this pointer is used
// this code is illegal: we may not explicitly define the this pointer ourselves
// note that this is a pointer to const because isbn is a const member
std::string Sales_data::isbn(const Sales_data *const this)
{ return this->isbn; }
```

The fact that `this` is a pointer to `const` means that `const` member functions cannot change the object on which they are called. Thus, `isbn` may read but not write to the data members of the objects on which it is called.



Objects that are `const`, and references or pointers to `const` objects, may call only `const` member functions.

## Class Scope and Member Functions

Recall that a class is itself a scope (§ 2.6.1, p. 72). The definitions of the member functions of a class are nested inside the scope of the class itself. Hence, `isbn`'s use of the name `bookNo` is resolved as the data member defined inside `Sales_data`.

It is worth noting that `isbn` can use `bookNo` even though `bookNo` is defined *after* `isbn`. As we'll see in § 7.4.1 (p. 283), the compiler processes classes in two steps—the member declarations are compiled first, after which the member function bodies, if any, are processed. Thus, member function bodies may use other members of their class regardless of where in the class those members appear.

## Defining a Member Function outside the Class

As with any other function, when we define a member function outside the class body, the member's definition must match its declaration. That is, the return type, parameter list, and name must match the declaration in the class body. If the member was declared as a `const` member function, then the definition must also specify `const` after the parameter list. The name of a member defined outside the class must include the name of the class of which it is a member:

```
double Sales_data::avg_price() const {
    if (units_sold)
        return revenue/units_sold;
    else
        return 0;
}
```

The function name, `Sales_data::avg_price`, uses the scope operator (§ 1.2, p. 8) to say that we are defining the function named `avg_price` that is declared in the scope of the `Sales_data` class. Once the compiler sees the function name, the rest of the code is interpreted as being inside the scope of the class. Thus, when `avg_price` refers to `revenue` and `units_sold`, it is implicitly referring to the members of `Sales_data`.

## Defining a Function to Return “This” Object

The `combine` function is intended to act like the compound assignment operator, `+=`. The object on which this function is called represents the left-hand operand of the assignment. The right-hand operand is passed as an explicit argument:

```
Sales_data& Sales_data::combine(const Sales_data &rhs)
{
    units_sold += rhs.units_sold; // add the members of rhs into
    revenue += rhs.revenue;      // the members of "this" object
    return *this; // return the object on which the function was called
}
```

When our transaction-processing program calls

```
total.combine(trans); // update the running total
```

the address of `total` is bound to the implicit `this` parameter and `rhs` is bound to `trans`. Thus, when `combine` executes

```
units_sold += rhs.units_sold; // add the members of rhs into
```

the effect is to add `total.units_sold` and `trans.units_sold`, storing the result back into `total.units_sold`.

The interesting part about this function is its return type and the `return` statement. Ordinarily, when we define a function that operates like a built-in operator, our function should mimic the behavior of that operator. The built-in assignment operators return their left-hand operand as an lvalue (§ 4.4, p. 144). To return an lvalue, our `combine` function must return a reference (§ 6.3.2, p. 226). Because the left-hand operand is a `Sales_data` object, the return type is `Sales_data&`.

As we've seen, we do not need to use the implicit `this` pointer to access the members of the object on which a member function is executing. However, we do need to use `this` to access the object as a whole:

```
return *this; // return the object on which the function was called
```

Here the `return` statement dereferences `this` to obtain the object on which the function is executing. That is, for the call above, we return a reference to `total`.

## EXERCISES SECTION 7.1.2

**Exercise 7.2:** Add the `combine` and `isbn` members to the `Sales_data` class you wrote for the exercises in § 2.6.2 (p. 76).

**Exercise 7.3:** Revise your transaction-processing program from § 7.1.1 (p. 256) to use these members.

**Exercise 7.4:** Write a class named `Person` that represents the name and address of a person. Use a `string` to hold each of these elements. Subsequent exercises will incrementally add features to this class.

**Exercise 7.5:** Provide operations in your `Person` class to return the name and address. Should these functions be `const`? Explain your choice.



### 7.1.3 Defining Nonmember Class-Related Functions

Class authors often define auxiliary functions, such as our `add`, `read`, and `print` functions. Although such functions define operations that are conceptually part of the interface of the class, they are not part of the class itself.

We define nonmember functions as we would any other function. As with any other function, we normally separate the declaration of the function from its

definition (§ 6.1.2, p. 206). Functions that are conceptually part of a class, but not defined inside the class, are typically declared (but not defined) in the same header as the class itself. That way users need to include only one file to use any part of the interface.



Ordinarily, nonmember functions that are part of the interface of a class should be declared in the same header as the class itself.

## Defining the `read` and `print` Functions

The `read` and `print` functions do the same job as the code in § 2.6.2 (p. 75) and not surprisingly, the bodies of our functions look a lot like the code presented there:

```
// input transactions contain ISBN, number of copies sold, and sales price
istream &read(istream &is, Sales_data &item)
{
    double price = 0;
    is >> item.bookNo >> item.units_sold >> price;
    item.revenue = price * item.units_sold;
    return is;
}
ostream &print(ostream &os, const Sales_data &item)
{
    os << item.isbn() << " " << item.units_sold << " "
        << item.revenue << " " << item.avg_price();
    return os;
}
```

The `read` function reads data from the given stream into the given object. The `print` function prints the contents of the given object on the given stream.

However, there are two points worth noting about these functions. First, both `read` and `print` take a reference to their respective IO class types. The IO classes are types that cannot be copied, so we may only pass them by reference (§ 6.2.2, p. 210). Moreover, reading or writing to a stream changes that stream, so both functions take ordinary references, not references to `const`.

The second thing to note is that `print` does not print a newline. Ordinarily, functions that do output should do minimal formatting. That way user code can decide whether the newline is needed.

## Defining the `add` Function

The `add` function takes two `Sales_data` objects and returns a new `Sales_data` representing their sum:

```
Sales_data add(const Sales_data &lhs, const Sales_data &rhs)
{
    Sales_data sum = lhs; // copy data members from lhs into sum
    sum.combine(rhs); // add data members from rhs into sum
    return sum;
}
```

In the body of the function we define a new `Sales_data` object named `sum` to hold the sum of our two transactions. We initialize `sum` as a copy of `lhs`. By default, copying a class object copies that object's members. After the copy, the `bookNo`, `units_sold`, and `revenue` members of `sum` will have the same values as those in `lhs`. Next we call `combine` to add the `units_sold` and `revenue` members of `rhs` into `sum`. When we're done, we return a copy of `sum`.

## EXERCISES SECTION 7.1.3

**Exercise 7.6:** Define your own versions of the `add`, `read`, and `print` functions.

**Exercise 7.7:** Rewrite the transaction-processing program you wrote for the exercises in § 7.1.2 (p. 260) to use these new functions.

**Exercise 7.8:** Why does `read` define its `Sales_data` parameter as a plain reference and `print` define its parameter as a reference to `const`?

**Exercise 7.9:** Add operations to `read` and `print Person` objects to the code you wrote for the exercises in § 7.1.2 (p. 260).

**Exercise 7.10:** What does the condition in the following `if` statement do?

```
if (read(read(cin, data1), data2))
```



### 7.1.4 Constructors

Each class defines how objects of its type can be initialized. Classes control object initialization by defining one or more special member functions known as **constructors**. The job of a constructor is to initialize the data members of a class object. A constructor is run whenever an object of a class type is created.

In this section, we'll introduce the basics of how to define a constructor. Constructors are a surprisingly complex topic. Indeed, we'll have more to say about constructors in § 7.5 (p. 288), § 15.7 (p. 622), and § 18.1.3 (p. 777), and in Chapter 13.

Constructors have the same name as the class. Unlike other functions, constructors have no return type. Like other functions, constructors have a (possibly empty) parameter list and a (possibly empty) function body. A class can have multiple constructors. Like any other overloaded function (§ 6.4, p. 230), the constructors must differ from each other in the number or types of their parameters.

Unlike other member functions, constructors may not be declared as `const` (§ 7.1.2, p. 258). When we create a `const` object of a class type, the object does not assume its “constness” until after the constructor completes the object's initialization. Thus, constructors can write to `const` objects during their construction.



### The Synthesized Default Constructor

Our `Sales_data` class does not define any constructors, yet the programs we've written that use `Sales_data` objects compile and run correctly. As an example, the program on page 255 defined two objects:

```
Sales_data total;      // variable to hold the running sum  
Sales_data trans;     // variable to hold data for the next transaction
```

The question naturally arises: How are `total` and `trans` initialized?

We did not supply an initializer for these objects, so we know that they are default initialized (§ 2.2.1, p. 43). Classes control default initialization by defining a special constructor, known as the **default constructor**. The default constructor is one that takes no arguments.

As we'll see the default constructor is special in various ways, one of which is that if our class does not *explicitly* define any constructors, the compiler will *implicitly* define the default constructor for us.

The compiler-generated constructor is known as the **synthesized default constructor**. For most classes, this synthesized constructor initializes each data member of the class as follows:

- If there is an in-class initializer (§ 2.6.1, p. 73), use it to initialize the member.
- Otherwise, default-initialize (§ 2.2.1, p. 43) the member.

Because `Sales_data` provides initializers for `units_sold` and `revenue`, the synthesized default constructor uses those values to initialize those members. It default initializes `bookNo` to the empty string.

## Some Classes Cannot Rely on the Synthesized Default Constructor

Only fairly simple classes—such as the current definition of `Sales_data`—can rely on the synthesized default constructor. The most common reason that a class must define its own default constructor is that the compiler generates the default for us *only if we do not define any other constructors for the class*. If we define any constructors, the class will not have a default constructor unless we define that constructor ourselves. The basis for this rule is that if a class requires control to initialize an object in one case, then the class is likely to require control in all cases.



The compiler generates a default constructor automatically only if a class declares *no* constructors.

A second reason to define the default constructor is that for some classes, the synthesized default constructor does the wrong thing. Remember that objects of built-in or compound type (such as arrays and pointers) that are defined inside a block have undefined value when they are default initialized (§ 2.2.1, p. 43). The same rule applies to members of built-in type that are default initialized. Therefore, classes that have members of built-in or compound type should ordinarily either initialize those members inside the class or define their own version of the default constructor. Otherwise, users could create objects with members that have undefined value.



Classes that have members of built-in or compound type usually should rely on the synthesized default constructor *only* if all such members have in-class initializers.

A third reason that some classes must define their own default constructor is that sometimes the compiler is unable to synthesize one. For example, if a class has a member that has a class type, and that class doesn't have a default constructor, then the compiler can't initialize that member. For such classes, we must define our own version of the default constructor. Otherwise, the class will not have a usable default constructor. We'll see in § 13.1.6 (p. 508) additional circumstances that prevent the compiler from generating an appropriate default constructor.

## Defining the Sales\_data Constructors

For our `Sales_data` class we'll define four constructors with the following parameters:

- An `istream&` from which to read a transaction.
- A `const string&` representing an ISBN, an `unsigned` representing the count of how many books were sold, and a `double` representing the price at which the books sold.
- A `const string&` representing an ISBN. This constructor will use default values for the other members.
- An empty parameter list (i.e., the default constructor) which as we've just seen we must define because we have defined other constructors.

Adding these members to our class, we now have

```
struct Sales_data {
    // constructors added
    Sales_data() = default;
    Sales_data(const std::string &s): bookNo(s) { }
    Sales_data(const std::string &s, unsigned n, double p):
        bookNo(s), units_sold(n), revenue(p*n) { }
    Sales_data(std::istream &);

    // other members as before
    std::string isbn() const { return bookNo; }
    Sales_data& combine(const Sales_data&);
    double avg_price() const;
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
};
```

## What = default Means

We'll start by explaining the default constructor:

```
Sales_data() = default;
```

First, note that this constructor defines the default constructor because it takes no arguments. We are defining this constructor *only* because we want to provide other constructors as well as the default constructor. We want this constructor to do exactly the same work as the synthesized version we had been using.

Under the new standard, if we want the default behavior, we can ask the compiler to generate the constructor for us by writing `= default` after the parameter list. The `= default` can appear with the declaration inside the class body or on the definition outside the class body. Like any other function, if the `= default` appears inside the class body, the default constructor will be inlined; if it appears on the definition outside the class, the member will not be inlined by default.



The default constructor works for `Sales_data` only because we provide initializers for the data members with built-in type. If your compiler does not support in-class initializers, your default constructor should use the constructor initializer list (described immediately following) to initialize every member of the class.

C++  
11

## Constructor Initializer List

Next we'll look at the other two constructors that were defined inside the class:

```
Sales_data(const std::string &s): bookNo(s) { }
Sales_data(const std::string &s, unsigned n, double p):
    bookNo(s), units_sold(n), revenue(p*n) { }
```

The new parts in these definitions are the colon and the code between it and the curly braces that define the (empty) function bodies. This new part is a **constructor initializer list**, which specifies initial values for one or more data members of the object being created. The constructor initializer is a list of member names, each of which is followed by that member's initial value in parentheses (or inside curly braces). Multiple member initializations are separated by commas.

The constructor that has three parameters uses its first two parameters to initialize the `bookNo` and `units_sold` members. The initializer for `revenue` is calculated by multiplying the number of books sold by the price per book.

The constructor that has a single `string` parameter uses that `string` to initialize `bookNo` but does not explicitly initialize the `units_sold` and `revenue` members. When a member is omitted from the constructor initializer list, it is implicitly initialized using the same process as is used by the synthesized default constructor. In this case, those members are initialized by the in-class initializers. Thus, the constructor that takes a `string` is equivalent to

```
// has the same behavior as the original constructor defined above
Sales_data(const std::string &s):
    bookNo(s), units_sold(0), revenue(0){ }
```

It is usually best for a constructor to use an in-class initializer if one exists and gives the member the correct value. On the other hand, if your compiler does not yet support in-class initializers, then every constructor should explicitly initialize every member of built-in type.

Best  
Practices

Constructors should not override in-class initializers except to use a different initial value. If you can't use in-class initializers, each constructor should explicitly initialize every member of built-in type.

It is worth noting that both constructors have empty function bodies. The only work these constructors need to do is give the data members their values. If there is no further work, then the function body is empty.

## Defining a Constructor outside the Class Body

Unlike our other constructors, the constructor that takes an `istream` does have work to do. Inside its function body, this constructor calls `read` to give the data members new values:

```
Sales_data::Sales_data(std::istream &is)
{
    read(is, *this); // read will read a transaction from is into this object
}
```

Constructors have no return type, so this definition starts with the name of the function we are defining. As with any other member function, when we define a constructor outside of the class body, we must specify the class of which the constructor is a member. Thus, `Sales_data::Sales_data` says that we're defining the `Sales_data` member named `Sales_data`. This member is a constructor because it has the same name as its class.

In this constructor there is no constructor initializer list, although technically speaking, it would be more correct to say that the constructor initializer list is empty. Even though the constructor initializer list is empty, the members of this object are still initialized before the constructor body is executed.

Members that do not appear in the constructor initializer list are initialized by the corresponding in-class initializer (if there is one) or are default initialized. For `Sales_data` that means that when the function body starts executing, `bookNo` will be the empty string, and `units_sold` and `revenue` will both be 0.

To understand the call to `read`, remember that `read`'s second parameter is a reference to a `Sales_data` object. In § 7.1.2 (p. 259), we noted that we use `this` to access the object as a whole, rather than a member of the object. In this case, we use `*this` to pass “`this`” object as an argument to the `read` function.

### EXERCISES SECTION 7.1.4

**Exercise 7.11:** Add constructors to your `Sales_data` class and write a program to use each of the constructors.

**Exercise 7.12:** Move the definition of the `Sales_data` constructor that takes an `istream` into the body of the `Sales_data` class.

**Exercise 7.13:** Rewrite the program from page 255 to use the `istream` constructor.

**Exercise 7.14:** Write a version of the default constructor that explicitly initializes the members to the values we have provided as in-class initializers.

**Exercise 7.15:** Add appropriate constructors to your `Person` class.

## 7.1.5 Copy, Assignment, and Destruction



In addition to defining how objects of the class type are initialized, classes also control what happens when we copy, assign, or destroy objects of the class type. Objects are copied in several contexts, such as when we initialize a variable or when we pass or return an object by value (§ 6.2.1, p. 209, and § 6.3.2, p. 224). Objects are assigned when we use the assignment operator (§ 4.4, p. 144). Objects are destroyed when they cease to exist, such as when a local object is destroyed on exit from the block in which it was created (§ 6.1.1, p. 204). Objects stored in a vector (or an array) are destroyed when that vector (or array) is destroyed.

If we do not define these operations, the compiler will synthesize them for us. Ordinarily, the versions that the compiler generates for us execute by copying, assigning, or destroying each member of the object. For example, in our bookstore program in § 7.1.1 (p. 255), when the compiler executes this assignment

```
total = trans; // process the next book
```

it executes as if we had written

```
// default assignment for Sales_data is equivalent to:  
total.bookNo = trans.bookNo;  
total.units_sold = trans.units_sold;  
total.revenue = trans.revenue;
```

We'll show how we can define our own versions of these operations in Chapter 13.

## Some Classes Cannot Rely on the Synthesized Versions



Although the compiler will synthesize the copy, assignment, and destruction operations for us, it is important to understand that for some classes the default versions do not behave appropriately. In particular, the synthesized versions are unlikely to work correctly for classes that allocate resources that reside outside the class objects themselves. As one example, in Chapter 12 we'll see how C++ programs allocate and manage dynamic memory. As we'll see in § 13.1.4 (p. 504), classes that manage dynamic memory, generally cannot rely on the synthesized versions of these operations.

However, it is worth noting that many classes that need dynamic memory can (and generally should) use a `vector` or a `string` to manage the necessary storage. Classes that use `vectors` and `strings` avoid the complexities involved in allocating and deallocating memory.

Moreover, the synthesized versions for copy, assignment, and destruction work correctly for classes that have `vector` or `string` members. When we copy or assign an object that has a `vector` member, the `vector` class takes care of copying or assigning the elements in that member. When the object is destroyed, the `vector` member is destroyed, which in turn destroys the elements in the `vector`. Similarly for `strings`.



Until you know how to define the operations covered in Chapter 13, the resources your classes allocate should be stored directly as data members of the class.



## 7.2 Access Control and Encapsulation

At this point, we have defined an interface for our class; but nothing forces users to use that interface. Our class is not yet encapsulated—users can reach inside a `Sales_data` object and meddle with its implementation. In C++ we use **access specifiers** to enforce encapsulation:

- Members defined after a `public` specifier are accessible to all parts of the program. The `public` members define the interface to the class.
- Members defined after a `private` specifier are accessible to the member functions of the class but are not accessible to code that uses the class. The `private` sections encapsulate (i.e., hide) the implementation.

Redefining `Sales_data` once again, we now have

```
class Sales_data {
public:           // access specifier added
    Sales_data() = default;
    Sales_data(const std::string &s, unsigned n, double p):
        bookNo(s), units_sold(n), revenue(p*n) { }
    Sales_data(const std::string &s): bookNo(s) { }
    Sales_data(std::istream&); 
    std::string isbn() const { return bookNo; }
    Sales_data &combine(const Sales_data&);
private:          // access specifier added
    double avg_price() const
    { return units_sold ? revenue/units_sold : 0; }
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
};
```

The constructors and member functions that are part of the interface (e.g., `isbn` and `combine`) follow the `public` specifier; the data members and the functions that are part of the implementation follow the `private` specifier.

A class may contain zero or more access specifiers, and there are no restrictions on how often an access specifier may appear. Each access specifier specifies the access level of the succeeding members. The specified access level remains in effect until the next access specifier or the end of the class body.

### Using the `class` or `struct` Keyword

We also made another, more subtle, change: We used the `class keyword` rather than `struct` to open the class definition. This change is strictly stylistic; we can define a class type using either keyword. The only difference between `struct` and `class` is the default access level.

A class may define members before the first access specifier. Access to such members depends on how the class is defined. If we use the `struct` keyword, the members defined before the first access specifier are `public`; if we use `class`, then the members are `private`.

As a matter of programming style, when we define a class intending for all of its members to be `public`, we use `struct`. If we intend to have `private` members, then we use `class`.



The *only* difference between using `class` and using `struct` to define a class is the default access level.

## EXERCISES SECTION 7.2

**Exercise 7.16:** What, if any, are the constraints on where and how often an access specifier may appear inside a class definition? What kinds of members should be defined after a `public` specifier? What kinds should be `private`?

**Exercise 7.17:** What, if any, are the differences between using `class` or `struct`?

**Exercise 7.18:** What is encapsulation? Why is it useful?

**Exercise 7.19:** Indicate which members of your `Person` class you would declare as `public` and which you would declare as `private`. Explain your choice.

### 7.2.1 Friends



Now that the data members of `Sales_data` are `private`, our `read`, `print`, and `add` functions will no longer compile. The problem is that although these functions are part of the `Sales_data` interface, they are not members of the class.

A class can allow another class or function to access its nonpublic members by making that class or function a **friend**. A class makes a function its friend by including a declaration for that function preceded by the keyword `friend`:

```
class Sales_data {  
    // friend declarations for nonmember Sales_data operations added  
    friend Sales_data add(const Sales_data&, const Sales_data&);  
    friend std::istream &read(std::istream&, Sales_data&);  
    friend std::ostream &print(std::ostream&, const Sales_data&);  
    // other members and access specifiers as before  
public:  
    Sales_data() = default;  
    Sales_data(const std::string &s, unsigned n, double p):  
        bookNo(s), units_sold(n), revenue(p*n) { }  
    Sales_data(const std::string &s): bookNo(s) { }  
    Sales_data(std::istream&);  
    std::string isbn() const { return bookNo; }  
    Sales_data &combine(const Sales_data&);  
private:  
    std::string bookNo;  
    unsigned units_sold = 0;  
    double revenue = 0.0;  
};
```

```
// declarations for nonmember parts of the Sales_data interface
Sales_data add(const Sales_data&, const Sales_data&);
std::istream &read(std::istream&, Sales_data&);
std::ostream &print(std::ostream&, const Sales_data&);
```

Friend declarations may appear only inside a class definition; they may appear anywhere in the class. Friends are not members of the class and are not affected by the access control of the section in which they are declared. We'll have more to say about friendship in § 7.3.4 (p. 279).



Ordinarily it is a good idea to group friend declarations together at the beginning or end of the class definition.

### KEY CONCEPT: BENEFITS OF ENCAPSULATION

Encapsulation provides two important advantages:

- User code cannot inadvertently corrupt the state of an encapsulated object.
- The implementation of an encapsulated class can change over time without requiring changes in user-level code.

By defining data members as `private`, the class author is free to make changes in the data. If the implementation changes, only the class code needs to be examined to see what effect the change may have. User code needs to change only when the interface changes. If the data are `public`, then any code that used the old data members might be broken. It would be necessary to locate and rewrite any code that relied on the old representation before the program could be used again.

Another advantage of making data members `private` is that the data are protected from mistakes that users might introduce. If there is a bug that corrupts an object's state, the places to look for the bug are localized: Only code that is part of the implementation could be responsible for the error. The search for the mistake is limited, greatly easing the problems of maintenance and program correctness.



Although user code need not change when a class definition changes, the source files that use a class must be recompiled any time the class changes.



### Declarations for Friends

A friend declaration only specifies access. It is not a general declaration of the function. If we want users of the class to be able to call a friend function, then we must also declare the function separately from the friend declaration.

To make a friend visible to users of the class, we usually declare each friend (outside the class) in the same header as the class itself. Thus, our `Sales_data` header should provide separate declarations (aside from the friend declarations inside the class body) for `read`, `print`, and `add`.



Many compilers do not enforce the rule that friend functions must be declared *outside* the class before they can be used.

Some compilers allow calls to a `friend` function when there is no ordinary declaration for that function. Even if your compiler allows such calls, it is a good idea to provide separate declarations for `friends`. That way you won't have to change your code if you use a compiler that enforces this rule.

### EXERCISES SECTION 7.2.1

**Exercise 7.20:** When are friends useful? Discuss the pros and cons of using friends.

**Exercise 7.21:** Update your `Sales_data` class to hide its implementation. The programs you've written to use `Sales_data` operations should still continue to work. Recompile those programs with your new class definition to verify that they still work.

**Exercise 7.22:** Update your `Person` class to hide its implementation.

## 7.3 Additional Class Features

The `Sales_data` class is pretty simple, yet it allowed us to explore quite a bit of the language support for classes. In this section, we'll cover some additional class-related features that `Sales_data` doesn't need to use. These features include type members, in-class initializers for members of class type, mutable data members, inline member functions, returning `*this` from a member function, more about how we define and use class types, and class friendship.

### 7.3.1 Class Members Revisited

To explore several of these additional features, we'll define a pair of cooperating classes named `Screen` and `Window_mngr`.

#### Defining a Type Member

A `Screen` represents a window on a display. Each `Screen` has a `string` member that holds the `Screen`'s contents, and three `string::size_type` members that represent the position of the cursor, and the height and width of the screen.

In addition to defining data and function members, a class can define its own local names for types. Type names defined by a class are subject to the same access controls as any other member and may be either `public` or `private`:

```
class Screen {
public:
    typedef std::string::size_type pos;
private:
    pos cursor = 0;
    pos height = 0, width = 0;
    std::string contents;
};
```

We defined `pos` in the public part of `Screen` because we want users to use that name. Users of `Screen` shouldn't know that `Screen` uses a `string` to hold its data. By defining `pos` as a public member, we can hide this detail of how `Screen` is implemented.

There are two points to note about the declaration of `pos`. First, although we used a `typedef` (§ 2.5.1, p. 67), we can equivalently use a type alias (§ 2.5.1, p. 68):

```
class Screen {
public:
    // alternative way to declare a type member using a type alias
    using pos = std::string::size_type;
    // other members as before
};
```

The second point is that, for reasons we'll explain in § 7.4.1 (p. 284), unlike ordinary members, members that define types must appear before they are used. As a result, type members usually appear at the beginning of the class.

## Member Functions of class `Screen`

To make our class more useful, we'll add a constructor that will let users define the size and contents of the screen, along with members to move the cursor and to get the character at a given location:

```
class Screen {
public:
    typedef std::string::size_type pos;
    Screen() = default; // needed because Screen has another constructor
    // cursor initialized to 0 by its in-class initializer
    Screen(pos ht, pos wd, char c): height(ht), width(wd),
                                    contents(ht * wd, c) { }
    char get() const           // get the character at the cursor
    { return contents[cursor]; } // implicitly inline
    inline char get(pos ht, pos wd) const; // explicitly inline
    Screen &move(pos r, pos c); // can be made inline later
private:
    pos cursor = 0;
    pos height = 0, width = 0;
    std::string contents;
};
```

Because we have provided a constructor, the compiler will not automatically generate a default constructor for us. If our class is to have a default constructor, we must say so explicitly. In this case, we use `= default` to ask the compiler to synthesize the default constructor's definition for us (§ 7.1.4, p. 264).

It's also worth noting that our second constructor (that takes three arguments) implicitly uses the in-class initializer for the `cursor` member (§ 7.1.4, p. 266). If our class did not have an in-class initializer for `cursor`, we would have explicitly initialized `cursor` along with the other members.

## Making Members `inline`

Classes often have small functions that can benefit from being inlined. As we've seen, member functions defined inside the class are automatically `inline` (§ 6.5.2, p. 238). Thus, `Screen`'s constructors and the version of `get` that returns the character denoted by the cursor are `inline` by default.

We can explicitly declare a member function as `inline` as part of its declaration inside the class body. Alternatively, we can specify `inline` on the function definition that appears outside the class body:

```
inline                  // we can specify inline on the definition
Screen &Screen::move(pos r, pos c)
{
    pos row = r * width; // compute the row location
    cursor = row + c;   // move cursor to the column within that row
    return *this;        // return this object as an lvalue
}
char Screen::get(pos r, pos c) const // declared as inline in the class
{
    pos row = r * width;      // compute row location
    return contents[row + c]; // return character at the given column
}
```

Although we are not required to do so, it is legal to specify `inline` on both the declaration and the definition. However, specifying `inline` only on the definition outside the class can make the class easier to read.



For the same reasons that we define `inline` functions in headers (§ 6.5.2, p. 240), `inline` member functions should be defined in the same header as the corresponding class definition.

## Overloading Member Functions

As with nonmember functions, member functions may be overloaded (§ 6.4, p. 230) so long as the functions differ by the number and/or types of parameters. The same function-matching (§ 6.4, p. 233) process is used for calls to member functions as for nonmember functions.

For example, our `Screen` class defined two versions of `get`. One version returns the character currently denoted by the cursor; the other returns the character at a given position specified by its row and column. The compiler uses the number of arguments to determine which version to run:

```
Screen myscreen;
char ch = myscreen.get(); // calls Screen::get()
ch = myscreen.get(0,0); // calls Screen::get(pos, pos)
```

## `mutable` Data Members

It sometimes (but not very often) happens that a class has a data member that we want to be able to modify, even inside a `const` member function. We indicate such members by including the `mutable` keyword in their declaration.

A **mutable data member** is never `const`, even when it is a member of a `const` object. Accordingly, a `const` member function may change a `mutable` member. As an example, we'll give `Screen` a `mutable` member named `access_ctr`, which we'll use to track how often each `Screen` member function is called:

```
class Screen {
public:
    void some_member() const;
private:
    mutable size_t access_ctr; // may change even in a const object
    // other members as before
};
void Screen::some_member() const
{
    ++access_ctr; // keep a count of the calls to any member function
    // whatever other work this member needs to do
}
```

Despite the fact that `some_member` is a `const` member function, it can change the value of `access_ctr`. That member is a `mutable` member, so any member function, including `const` functions, can change its value.

## Initializers for Data Members of Class Type

In addition to defining the `Screen` class, we'll define a window manager class that represents a collection of `Screens` on a given display. This class will have a `vector` of `Screens` in which each element represents a particular `Screen`. By default, we'd like our `Window_mngr` class to start up with a single, default-initialized `Screen`. Under the new standard, the best way to specify this default value is as an in-class initializer (§ 2.6.1, p. 73):

```
class Window_mngr {
private:
    // Screens this Window_mngr is tracking
    // by default, a Window_mngr has one standard sized blank Screen
    std::vector<Screen> screens{Screen(24, 80, ' ')};
};
```

When we initialize a member of class type, we are supplying arguments to a constructor of that member's type. In this case, we list initialize our `vector` member (§ 3.3.1, p. 98) with a single element initializer. That initializer contains a `Screen` value that is passed to the `vector<Screen>` constructor to create a one-element vector. That value is created by the `Screen` constructor that takes two size parameters and a character to create a blank screen of the given size.

As we've seen, in-class initializers must use either the `=` form of initialization (which we used when we initialized the the data members of `Screen`) or the direct form of initialization using curly braces (as we do for `screens`).



When we provide an in-class initializer, we must do so following an `=` sign or inside braces.

## EXERCISES SECTION 7.3.1

**Exercise 7.23:** Write your own version of the Screen class.

**Exercise 7.24:** Give your Screen class three constructors: a default constructor; a constructor that takes values for height and width and initializes the contents to hold the given number of blanks; and a constructor that takes values for height, width, and a character to use as the contents of the screen.

**Exercise 7.25:** Can Screen safely rely on the default versions of copy and assignment? If so, why? If not, why not?

**Exercise 7.26:** Define Sales\_data::avg\_price as an inline function.

### 7.3.2 Functions That Return `*this`



Next we'll add functions to set the character at the cursor or at a given location:

```
class Screen {
public:
    Screen &set(char);
    Screen &set(pos, pos, char);
    // other members as before
};

inline Screen &Screen::set(char c)
{
    contents[cursor] = c; // set the new value at the current cursor location
    return *this;           // return this object as an lvalue
}

inline Screen &Screen::set(pos r, pos col, char ch)
{
    contents[r*width + col] = ch; // set specified location to given value
    return *this;               // return this object as an lvalue
}
```

Like the move operation, our set members return a reference to the object on which they are called (§ 7.1.2, p. 259). Functions that return a reference are lvalues (§ 6.3.2, p. 226), which means that they return the object itself, not a copy of the object. If we concatenate a sequence of these actions into a single expression:

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

these operations will execute on the same object. In this expression, we first move the cursor inside myScreen and then set a character in myScreen's contents member. That is, this statement is equivalent to

```
myScreen.move(4,0);
myScreen.set('#');
```

Had we defined move and set to return Screen, rather than Screen&, this statement would execute quite differently. In this case it would be equivalent to:

```
// if move returns Screen not Screen&
Screen temp = myScreen.move(4,0); // the return value would be copied
temp.set('#'); // the contents inside myScreen would be unchanged
```

If `move` had a nonreference return type, then the return value of `move` would be a copy of `*this` (§ 6.3.2, p. 224). The call to `set` would change the temporary copy, not `myScreen`.

## Returning `*this` from a `const` Member Function

Next, we'll add an operation, which we'll name `display`, to print the contents of the `Screen`. We'd like to be able to include this operation in a sequence of `set` and `move` operations. Therefore, like `set` and `move`, our `display` function will return a reference to the object on which it executes.

Logically, displaying a `Screen` doesn't change the object, so we should make `display` a `const` member. If `display` is a `const` member, then `this` is a pointer to `const` and `*this` is a `const` object. Hence, the return type of `display` must be `const Sales_data&`. However, if `display` returns a reference to `const`, we won't be able to embed `display` into a series of actions:

```
Screen myScreen;
// if display returns a const reference, the call to set is an error
myScreen.display(cout).set('*');
```

Even though `myScreen` is a nonconst object, the call to `set` won't compile. The problem is that the `const` version of `display` returns a reference to `const` and we cannot call `set` on a `const` object.



A `const` member function that returns `*this` as a reference should have a return type that is a reference to `const`.

## Overloading Based on `const`

We can overload a member function based on whether it is `const` for the same reasons that we can overload a function based on whether a pointer parameter points to `const` (§ 6.4, p. 232). The nonconst version will not be viable for `const` objects; we can only call `const` member functions on a `const` object. We can call either version on a nonconst object, but the nonconst version will be a better match.

In this example, we'll define a `private` member named `do_display` to do the actual work of printing the `Screen`. Each of the `display` operations will call this function and then return the object on which it is executing:

```
class Screen {
public:
    // display overloaded on whether the object is const or not
    Screen &display(std::ostream &os)
        { do_display(os); return *this; }
    const Screen &display(std::ostream &os) const
        { do_display(os); return *this; }
```

```
private:  
    // function to do the work of displaying a Screen  
    void do_display(std::ostream &os) const {os << contents;}  
    // other members as before  
};
```

As in any other context, when one member calls another the `this` pointer is passed implicitly. Thus, when `display` calls `do_display`, its own `this` pointer is implicitly passed to `do_display`. When the `nonconst` version of `display` calls `do_display`, its `this` pointer is implicitly converted from a pointer to `nonconst` to a pointer to `const` (§ 4.11.2, p. 162).

When `do_display` completes, the `display` functions each return the object on which they execute by dereferencing `this`. In the `nonconst` version, `this` points to a `nonconst` object, so that version of `display` returns an ordinary (`nonconst`) reference; the `const` member returns a reference to `const`.

When we call `display` on an object, whether that object is `const` determines which version of `display` is called:

```
Screen myScreen(5,3);  
const Screen blank(5, 3);  
myScreen.set('#').display(cout); // calls nonconst version  
blank.display(cout);           // calls const version
```

#### ADVICE: USE PRIVATE UTILITY FUNCTIONS FOR COMMON CODE

Some readers might be surprised that we bothered to define a separate `do_display` operation. After all, the calls to `do_display` aren't much simpler than the action done inside `do_display`. Why bother? We do so for several reasons:

- A general desire to avoid writing the same code in more than one place.
- We expect that the `display` operation will become more complicated as our class evolves. As the actions involved become more complicated, it makes more obvious sense to write those actions in one place, not two.
- It is likely that we might want to add debugging information to `do_display` during development that would be eliminated in the final product version of the code. It will be easier to do so if only one definition of `do_display` needs to be changed to add or remove the debugging code.
- There needn't be any overhead involved in this extra function call. We defined `do_display` inside the class body, so it is implicitly `inline`. Thus, there likely be no run-time overhead associating with calling `do_display`.

In practice, well-designed C++ programs tend to have lots of small functions such as `do_display` that are called to do the "real" work of some other set of functions.

### 7.3.3 Class Types

Every class defines a unique type. Two different classes define two different types even if they define the same members. For example:

## EXERCISES SECTION 7.3.2

**Exercise 7.27:** Add the move, set, and display operations to your version of Screen. Test your class by executing the following code:

```
Screen myScreen(5, 5, 'X');
myScreen.move(4,0).set('#').display(cout);
cout << "\n";
myScreen.display(cout);
cout << "\n";
```

**Exercise 7.28:** What would happen in the previous exercise if the return type of move, set, and display was Screen rather than Screen&?

**Exercise 7.29:** Revise your Screen class so that move, set, and display functions return Screen and check your prediction from the previous exercise.

**Exercise 7.30:** It is legal but redundant to refer to members through the this pointer. Discuss the pros and cons of explicitly using the this pointer to access members.

```
struct First {
    int memi;
    int getMem();
};

struct Second {
    int memi;
    int getMem();
};

First obj1;
Second obj2 = obj1; // error: obj1 and obj2 have different types
```



Even if two classes have exactly the same member list, they are different types. The members of each class are distinct from the members of any other class (or any other scope).

We can refer to a class type directly, by using the class name as a type name. Alternatively, we can use the class name following the keyword `class` or `struct`:

```
Sales_data item1;           // default-initialized object of type Sales_data
class Sales_data item1; // equivalent declaration
```

Both methods of referring to a class type are equivalent. The second method is inherited from C and is also valid in C++.

## Class Declarations

Just as we can declare a function apart from its definition (§ 6.1.2, p. 206), we can also declare a class without defining it:

```
class Screen; // declaration of the Screen class
```

This declaration, sometimes referred to as a **forward declaration**, introduces the name Screen into the program and indicates that Screen refers to a class type. After a declaration and before a definition is seen, the type Screen is an **incomplete type**—it's known that Screen is a class type but not known what members that type contains.

We can use an incomplete type in only limited ways: We can define pointers or references to such types, and we can declare (but not define) functions that use an incomplete type as a parameter or return type.

A class must be defined—not just declared—before we can write code that creates objects of that type. Otherwise, the compiler does not know how much storage such objects need. Similarly, the class must be defined before a reference or pointer is used to access a member of the type. After all, if the class has not been defined, the compiler can't know what members the class has.

With one exception that we'll describe in § 7.6 (p. 300), data members can be specified to be of a class type only if the class has been defined. The type must be complete because the compiler needs to know how much storage the data member requires. Because a class is not defined until its class body is complete, a class cannot have data members of its own type. However, a class is considered declared (but not yet defined) as soon as its class name has been seen. Therefore, a class can have data members that are pointers or references to its own type:

```
class Link_screen {  
    Screen window;  
    Link_screen *next;  
    Link_screen *prev;  
};
```

### EXERCISES SECTION 7.3.3

**Exercise 7.31:** Define a pair of classes X and Y, in which X has a pointer to Y, and Y has an object of type X.

## 7.3.4 Friendship Revisited

Our Sales\_data class defined three ordinary nonmember functions as friends (§ 7.2.1, p. 269). A class can also make another class its friend or it can declare specific member functions of another (previously defined) class as friends. In addition, a friend function can be defined inside the class body. Such functions are implicitly inline.

### Friendship between Classes

As an example of class friendship, our Window\_mgr class (§ 7.3.1, p. 274) will have members that will need access to the internal data of the Screen objects it manages. For example, let's assume that we want to add a member, named clear

to `Window_mgr` that will reset the contents of a particular `Screen` to all blanks. To do this job, `clear` needs to access the private data members of `Screen`. To allow this access, `Screen` can designate `Window_mgr` as its friend:

```
class Screen {
    // Window_mgr members can access the private parts of class Screen
    friend class Window_mgr;
    // ... rest of the Screen class
};
```

The member functions of a friend class can access all the members, including the nonpublic members, of the class granting friendship. Now that `Window_mgr` is a friend of `Screen`, we can write the `clear` member of `Window_mgr` as follows:

```
class Window_mgr {
public:
    // location ID for each screen on the window
    using ScreenIndex = std::vector<Screen>::size_type;
    // reset the Screen at the given position to all blanks
    void clear(ScreenIndex i);
private:
    std::vector<Screen> screens{Screen(24, 80, ' ')};
};

void Window_mgr::clear(ScreenIndex i)
{
    // s is a reference to the Screen we want to clear
    Screen &s = screens[i];
    // reset the contents of that Screen to all blanks
    s.contents = string(s.height * s.width, ' ');*/
}
```

We start by defining `s` as a reference to the `Screen` at position `i` in the `screens` vector. We then use the `height` and `width` members of that `Screen` to compute a new string that has the appropriate number of blank characters. We assign that string of blanks to the `contents` member.

If `clear` were not a friend of `Screen`, this code would not compile. The `clear` function would not be allowed to use the `height` `width`, or `contents` members of `Screen`. Because `Screen` grants friendship to `Window_mgr`, all the members of `Screen` are accessible to the functions in `Window_mgr`.

It is important to understand that friendship is not transitive. That is, if class `Window_mgr` has its own friends, those friends have no special access to `Screen`.



Each class controls which classes or functions are its friends.

## Making A Member Function a Friend

Rather than making the entire `Window_mgr` class a friend, `Screen` can instead specify that only the `clear` member is allowed access. When we declare a member function to be a friend, we must specify the class of which that function is a member:

```

class Screen {
    // Window_mgr::clear must have been declared before class Screen
    friend void Window_mgr::clear(ScreenIndex);
    // ... rest of the Screen class
};

```

Making a member function a friend requires careful structuring of our programs to accommodate interdependencies among the declarations and definitions. In this example, we must order our program as follows:

- First, define the `Window_mgr` class, which declares, but cannot define, `clear`. `Screen` must be declared before `clear` can use the members of `Screen`.
- Next, define class `Screen`, including a friend declaration for `clear`.
- Finally, define `clear`, which can now refer to the members in `Screen`.

## Overloaded Functions and Friendship

Although overloaded functions share a common name, they are still different functions. Therefore, a class must declare as a friend each function in a set of overloaded functions that it wishes to make a friend:

```

// overloaded storeOn functions
extern std::ostream& storeOn(std::ostream &, Screen &);
extern BitMap& storeOn(BitMap &, Screen &);

class Screen {
    // ostream version of storeOn may access the private parts of Screen objects
    friend std::ostream& storeOn(std::ostream &, Screen &);
    // ...
};

```

Class `Screen` makes the version of `storeOn` that takes an `ostream&` its friend. The version that takes a `BitMap&` has no special access to `Screen`.

## Friend Declarations and Scope

Classes and nonmember functions need not have been declared before they are used in a friend declaration. When a name first appears in a friend declaration, that name is implicitly *assumed* to be part of the surrounding scope. However, the friend itself is not actually declared in that scope (§ 7.2.1, p. 270).



Even if we define the function inside the class, we must still provide a declaration outside of the class itself to make that function visible. A declaration must exist even if we only call the friend from members of the friendship granting class:

```

struct X {
    friend void f() { /* friend function can be defined in the class body */ }
    x() { f(); } // error: no declaration for f
    void g();
    void h();
};

void X::g() { return f(); } // error: f hasn't been declared

```

```
void f(); // declares the function defined inside X
void X::h() { return f(); } // ok: declaration for f is now in scope
```

It is important to understand that a friend declaration affects access but is not a declaration in an ordinary sense.



Remember, some compilers do not enforce the lookup rules for friends (§ 7.2.1, p. 270).

## EXERCISES SECTION 7.3.4

**Exercise 7.32:** Define your own versions of `Screen` and `Window_mngr` in which `clear` is a member of `Window_mngr` and a friend of `Screen`.



## 7.4 Class Scope

Every class defines its own new scope. Outside the class scope, ordinary data and function members may be accessed only through an object, a reference, or a pointer using a member access operator (§ 4.6, p. 150). We access type members from the class using the scope operator `.`. In either case, the name that follows the operator must be a member of the associated class.

```
Screen::pos ht = 24, wd = 80; // use the pos type defined by Screen
Screen scr(ht, wd, ' ');
Screen *p = &scr;
char c = scr.get(); // fetches the get member from the object scr
c = p->get(); // fetches the get member from the object to which p points
```

### Scope and Members Defined outside the Class

The fact that a class is a scope explains why we must provide the class name as well as the function name when we define a member function outside its class (§ 7.1.2, p. 259). Outside of the class, the names of the members are hidden.

Once the class name is seen, the remainder of the definition—including the parameter list and the function body—is in the scope of the class. As a result, we can refer to other class members without qualification.

For example, recall the `clear` member of class `Window_mngr` (§ 7.3.4, p. 280). That function's parameter uses a type that is defined by `Window_mngr`:

```
void Window_mngr::clear(ScreenIndex i)
{
    Screen &s = screens[i];
    s.contents = string(s.height * s.width, '');
```

Because the compiler sees the parameter list after noting that we are in the scope of class `Window_mngr`, there is no need to specify that we want the `ScreenIndex`

that is defined by `window_mgr`. For the same reason, the use of `screens` in the function body refers to name declared inside class `Window_mgr`.

On the other hand, the return type of a function normally appears before the function's name. When a member function is defined outside the class body, any name used in the return type is outside the class scope. As a result, the return type must specify the class of which it is a member. For example, we might give `Window_mgr` a function, named `addScreen`, to add another screen to the display. This member will return a `ScreenIndex` value that the user can subsequently use to locate this `Screen`:

```
class Window_mgr {  
public:  
    // add a Screen to the window and returns its index  
    ScreenIndex addScreen(const Screen&);  
    // other members as before  
};  
// return type is seen before we're in the scope of Window_mgr  
Window_mgr::ScreenIndex  
Window_mgr::addScreen(const Screen &s)  
{  
    screens.push_back(s);  
    return screens.size() - 1;  
}
```

Because the return type appears before the name of the class is seen, it appears outside the scope of class `Window_mgr`. To use `ScreenIndex` for the return type, we must specify the class in which that type is defined.

## EXERCISES SECTION 7.4

**Exercise 7.33:** What would happen if we gave `Screen` a `size` member defined as follows? Fix any problems you identify.

```
pos Screen::size() const  
{  
    return height * width;  
}
```

### 7.4.1 Name Lookup and Class Scope

In the programs we've written so far, **name lookup** (the process of finding which declarations match the use of a name) has been relatively straightforward:

- First, look for a declaration of the name in the block in which the name was used. Only names declared before the use are considered.
- If the name isn't found, look in the enclosing scope(s).
- If no declaration is found, then the program is in error.



The way names are resolved inside member functions defined inside the class may seem to behave differently than these lookup rules. However, in this case, appearances are deceiving. Class definitions are processed in two phases:

- First, the member declarations are compiled.
- Function bodies are compiled only after the entire class has been seen.



Member function definitions are processed *after* the compiler processes all of the declarations in the class.

Classes are processed in this two-phase way to make it easier to organize class code. Because member function bodies are not processed until the entire class is seen, they can use any name defined inside the class. If function definitions were processed at the same time as the member declarations, then we would have to order the member functions so that they referred only to names already seen.

## Name Lookup for Class Member Declarations

This two-step process applies only to names used in the body of a member function. Names used in declarations, including names used for the return type and types in the parameter list, must be seen before they are used. If a member declaration uses a name that has not yet been seen inside the class, the compiler will look for that name in the scope(s) in which the class is defined. For example:

```
typedef double Money;
string bal;

class Account {
public:
    Money balance() { return bal; }
private:
    Money bal;
    // ...
};
```

When the compiler sees the declaration of the `balance` function, it will look for a declaration of `Money` in the `Account` class. The compiler considers only declarations inside `Account` that appear before the use of `Money`. Because no matching member is found, the compiler then looks for a declaration in the enclosing scope(s). In this example, the compiler will find the `typedef` of `Money`. That type will be used for the return type of the function `balance` and as the type for the data member `bal`. On the other hand, the function body of `balance` is processed only after the entire class is seen. Thus, the `return` inside that function returns the member named `bal`, not the `string` from the outer scope.

## Type Names Are Special

Ordinarily, an inner scope can redefine a name from an outer scope even if that name has already been used in the inner scope. However, in a class, if a member

uses a name from an outer scope and that name is a type, then the class may not subsequently redefine that name:

```
typedef double Money;
class Account {
public:
    Money balance() { return bal; } // uses Money from the outer scope
private:
    typedef double Money; // error: cannot redefine Money
    Money bal;
    // ...
};
```

It is worth noting that even though the definition of `Money` inside `Account` uses the same type as the definition in the outer scope, this code is still in error.

Although it is an error to redefine a type name, compilers are not required to diagnose this error. Some compilers will quietly accept such code, even though the program is in error.



Definitions of type names usually should appear at the beginning of a class. That way any member that uses that type will be seen after the type name has already been defined.

## Normal Block-Scope Name Lookup inside Member Definitions

A name used in the body of a member function is resolved as follows:

- First, look for a declaration of the name inside the member function. As usual, only declarations in the function body that precede the use of the name are considered.
- If the declaration is not found inside the member function, look for a declaration inside the class. All the members of the class are considered.
- If a declaration for the name is not found in the class, look for a declaration that is in scope before the member function definition.

Ordinarily, it is a bad idea to use the name of another member as the name for a parameter in a member function. However, in order to show how names are resolved, we'll violate that normal practice in our `dummy_fcn` function:

```
// note: this code is for illustration purposes only and reflects bad practice
// it is generally a bad idea to use the same name for a parameter and a member
int height; // defines a name subsequently used inside Screen
class Screen {
public:
    typedef std::string::size_type pos;
    void dummy_fcn(pos height) {
        cursor = width * height; // which height? the parameter
    }
}
```

```
private:
    pos cursor = 0;
    pos height = 0, width = 0;
};
```

When the compiler processes the multiplication expression inside `dummy_fcn`, it first looks for the names used in that expression in the scope of that function. A function's parameters are in the function's scope. Thus, the name `height`, used in the body of `dummy_fcn`, refers to this parameter declaration.

In this case, the `height` parameter hides the member named `height`. If we wanted to override the normal lookup rules, we can do so:

```
// bad practice: names local to member functions shouldn't hide member names
void Screen::dummy_fcn(pos height) {
    cursor = width * this->height; // member height
    // alternative way to indicate the member
    cursor = width * Screen::height; // member height
}
```



Even though the class member is hidden, it is still possible to use that member by qualifying the member's name with the name of its class or by using the `this` pointer explicitly.

A much better way to ensure that we get the member named `height` would be to give the parameter a different name:

```
// good practice: don't use a member name for a parameter or other local variable
void Screen::dummy_fcn(pos ht) {
    cursor = width * height; // member height
}
```

In this case, when the compiler looks for the name `height`, it won't be found inside `dummy_fcn`. The compiler next looks at all the declarations in `Screen`. Even though the declaration of `height` appears after its use inside `dummy_fcn`, the compiler resolves this use to the data member named `height`.

## After Class Scope, Look in the Surrounding Scope

If the compiler doesn't find the name in function or class scope, it looks for the name in the surrounding scope. In our example, the name `height` is defined in the outer scope before the definition of `Screen`. However, the object in the outer scope is hidden by our member named `height`. If we want the name from the outer scope, we can ask for it explicitly using the scope operator:

```
// bad practice: don't hide names that are needed from surrounding scopes
void Screen::dummy_fcn(pos height) {
    cursor = width * ::height; // which height? the global one
}
```



Even though the outer object is hidden, it is still possible to access that object by using the scope operator.

## Names Are Resolved Where They Appear within a File

When a member is defined outside its class, the third step of name lookup includes names declared in the scope of the member definition as well as those that appear in the scope of the class definition. For example:

```
int height;    // defines a name subsequently used inside Screen
class Screen {
public:
    typedef std::string::size_type pos;
    void setHeight(pos);
    pos height = 0; // hides the declaration of height in the outer scope
};
Screen::pos verify(Screen::pos);
void Screen::setHeight(pos var) {
    // var: refers to the parameter
    // height: refers to the class member
    // verify: refers to the global function
    height = verify(var);
}
```

Notice that the declaration of the global function `verify` is not visible before the definition of the class `Screen`. However, the third step of name lookup includes the scope in which the member definition appears. In this example, the declaration for `verify` appears before `setHeight` is defined and may, therefore, be used.

### EXERCISES SECTION 7.4.1

**Exercise 7.34:** What would happen if we put the `typedef pos` in the `Screen` class on page 285 as the last line in the class?

**Exercise 7.35:** Explain the following code, indicating which definition of `Type` or `initVal` is used for each use of those names. Say how you would fix any errors.

```
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();
    return val;
}
```

## 7.5 Constructors Revisited

Constructors are a crucial part of any C++ class. We covered the basics of constructors in § 7.1.4 (p. 262). In this section we'll cover some additional capabilities of constructors, and deepen our coverage of the material introduced earlier.



### 7.5.1 Constructor Initializer List

When we define variables, we typically initialize them immediately rather than defining them and then assigning to them:

```
string foo = "Hello World!"; // define and initialize
string bar;                // default initialized to the empty string
bar = "Hello World!";       // assign a new value to bar
```

Exactly the same distinction between initialization and assignment applies to the data members of objects. If we do not explicitly initialize a member in the constructor initializer list, that member is default initialized before the constructor body starts executing. For example:

```
// legal but sloppier way to write the Sales_data constructor: no constructor initializers
Sales_data::Sales_data(const string &s,
                      unsigned cnt, double price)
{
    bookNo = s;
    units_sold = cnt;
    revenue = cnt * price;
}
```

This version and our original definition on page 264 have the same effect: When the constructor finishes, the data members will hold the same values. The difference is that the original version *initializes* its data members, whereas this version *assigns* values to the data members. How significant this distinction is depends on the type of the data member.

### Constructor Initializers Are Sometimes Required

We can often, *but not always*, ignore the distinction between whether a member is initialized or assigned. Members that are `const` or references must be initialized. Similarly, members that are of a class type that does not define a default constructor also must be initialized. For example:

```
class ConstRef {
public:
    ConstRef(int ii);
private:
    int i;
    const int ci;
    int &ri;
};
```

Like any other `const` object or reference, the members `ci` and `ri` must be initialized. As a result, omitting a constructor initializer for these members is an error:

```
// error: ci and ri must be initialized
ConstRef::ConstRef(int ii)
{
    // assignments:
    i = ii; // ok
    ci = ii; // error: cannot assign to a const
    ri = i; // error: ri was never initialized
}
```

By the time the body of the constructor begins executing, initialization is complete. Our only chance to initialize `const` or reference data members is in the constructor initializer. The correct way to write this constructor is

```
// ok: explicitly initialize reference and const members
ConstRef::ConstRef(int ii): i(ii), ci(ii), ri(i) {}
```



We *must* use the constructor initializer list to provide values for members that are `const`, reference, or of a class type that does not have a default constructor.

#### ADVICE: USE CONSTRUCTOR INITIALIZERS

In many classes, the distinction between initialization and assignment is strictly a matter of low-level efficiency: A data member is initialized and then assigned when it could have been initialized directly.

More important than the efficiency issue is the fact that some data members must be initialized. By routinely using constructor initializers, you can avoid being surprised by compile-time errors when you have a class with a member that requires a constructor initializer.

## Order of Member Initialization

Not surprisingly, each member may be named only once in the constructor initializer. After all, what might it mean to give a member two initial values?

What may be more surprising is that the constructor initializer list specifies only the values used to initialize the members, not the order in which those initializations are performed.

Members are initialized in the order in which they appear in the class definition: The first member is initialized first, then the next, and so on. The order in which initializers appear in the constructor initializer list does not change the order of initialization.

The order of initialization often doesn't matter. However, if one member is initialized in terms of another, then the order in which members are initialized is crucially important.

As an example, consider the following class:

```
class X {
    int i;
    int j;
public:
    // undefined: i is initialized before j
    X(int val): j(val), i(j) { }
};
```

In this case, the constructor initializer makes it *appear* as if *j* is initialized with *val* and then *j* is used to initialize *i*. However, *i* is initialized first. The effect of this initializer is to initialize *i* with the undefined value of *j*!

Some compilers are kind enough to generate a warning if the data members are listed in the constructor initializer in a different order from the order in which the members are declared.



It is a good idea to write constructor initializers in the same order as the members are declared. Moreover, when possible, avoid using members to initialize other members.

If possible, it is a good idea write member initializers to use the constructor's parameters rather than another data member from the same object. That way we don't even have to think about the order of member initialization. For example, it would be better to write the constructor for *X* as

```
X(int val): i(val), j(val) { }
```

In this version, the order in which *i* and *j* are initialized doesn't matter.

## Default Arguments and Constructors

The actions of the *Sales\_data* default constructor are similar to those of the constructor that takes a single *string* argument. The only difference is that the constructor that takes a *string* argument uses that argument to initialize *bookNo*. The default constructor (implicitly) uses the *string* default constructor to initialize *bookNo*. We can rewrite these constructors as a single constructor with a default argument (§ 6.5.1, p. 236):

```
class Sales_data {
public:
    // defines the default constructor as well as one that takes a string argument
    Sales_data(std::string s = ""): bookNo(s) { }
    // remaining constructors unchanged
    Sales_data(std::string s, unsigned cnt, double rev):
        bookNo(s), units_sold(cnt), revenue(rev*cnt) { }
    Sales_data(std::istream &is) { read(is, *this); }
    // remaining members as before
};
```

This version of our class provides the same interface as our original on page 264. Both versions create the same object when given no arguments or when given a single *string* argument. Because we can call this constructor with no arguments, this constructor defines a default constructor for our class.



A constructor that supplies default arguments for all its parameters also defines the default constructor.

It is worth noting that we probably should not use default arguments with the `Sales_data` constructor that takes three arguments. If a user supplies a nonzero count for the number of books sold, we want to ensure that the user also supplies the price at which those books were sold.

### EXERCISES SECTION 7.5.1

**Exercise 7.36:** 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;  
};
```

**Exercise 7.37:** Using the version of `Sales_data` from this section, determine which constructor is used to initialize each of the following variables and list the values of the data members in each object:

```
Sales_data first_item(cin);  
  
int main() {  
    Sales_data next;  
    Sales_data last("9-999-99999-9");  
}
```

**Exercise 7.38:** We might want to supply `cin` as a default argument to the constructor that takes an `istream&`. Write the constructor declaration that uses `cin` as a default argument.

**Exercise 7.39:** 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?

**Exercise 7.40:** Choose one of the following abstractions (or an abstraction of your own choosing). Determine what data are needed in the class. Provide an appropriate set of constructors. Explain your decisions.

- (a) Book
- (b) Date
- (c) Employee
- (d) Vehicle
- (e) Object
- (f) Tree

### 7.5.2 Delegating Constructors

The new standard extends the use of constructor initializers to let us define so-called **delegating constructors**. A delegating constructor uses another constructor from its own class to perform its initialization. It is said to “delegate” some (or all) of its work to this other constructor.

Like any other constructor, a delegating constructor has a member initializer

list and a function body. In a delegating constructor, the member initializer list has a single entry that is the name of the class itself. Like other member initializers, the name of the class is followed by a parenthesized list of arguments. The argument list must match another constructor in the class.

As an example, we'll rewrite the `Sales_data` class to use delegating constructors as follows:

```
class Sales_data {  
public:  
    // nondelegating constructor initializes members from corresponding arguments  
    Sales_data(std::string s, unsigned cnt, double price):  
        bookNo(s), units_sold(cnt), revenue(cnt*price) {}  
    // remaining constructors all delegate to another constructor  
    Sales_data(): Sales_data("", 0, 0) {}  
    Sales_data(std::string s): Sales_data(s, 0, 0) {}  
    Sales_data(std::istream &is): Sales_data()  
        { read(is, *this); }  
    // other members as before  
};
```

In this version of `Sales_data`, all but one of the constructors delegate their work. The first constructor takes three arguments, uses those arguments to initialize the data members, and does no further work. In this version of the class, we define the default constructor to use the three-argument constructor to do its initialization. It too has no additional work, as indicated by the empty constructor body. The constructor that takes a `string` also delegates to the three-argument version.

The constructor that takes an `istream&` also delegates. It delegates to the default constructor, which in turn delegates to the three-argument constructor. Once those constructors complete their work, the body of the `istream&` constructor is run. Its constructor body calls `read` to read the given `istream`.

When a constructor delegates to another constructor, the constructor initializer list and function body of the delegated-to constructor are both executed. In `Sales_data`, the function bodies of the delegated-to constructors happen to be empty. Had the function bodies contained code, that code would be run before control returned to the function body of the delegating constructor.

## EXERCISES SECTION 7.5.2

**Exercise 7.41:** Rewrite your own version of the `Sales_data` class to use delegating constructors. Add a statement to the body of each of the constructors that prints a message whenever it is executed. Write declarations to construct a `Sales_data` object in every way possible. Study the output until you are certain you understand the order of execution among delegating constructors.

**Exercise 7.42:** For the class you wrote for exercise 7.40 in § 7.5.1 (p. 291), decide whether any of the constructors might use delegation. If so, write the delegating constructor(s) for your class. If not, look at the list of abstractions and choose one that you think would use a delegating constructor. Write the class definition for that abstraction.

### 7.5.3 The Role of the Default Constructor



The default constructor is used automatically whenever an object is default or value initialized. Default initialization happens

- When we define `nonstatic` variables (§ 2.2.1, p. 43) or arrays (§ 3.5.1, p. 114) at block scope without initializers
- When a class that itself has members of class type uses the synthesized default constructor (§ 7.1.4, p. 262)
- When members of class type are not explicitly initialized in a constructor initializer list (§ 7.1.4, p. 265)

Value initialization happens

- During array initialization when we provide fewer initializers than the size of the array (§ 3.5.1, p. 114)
- When we define a local static object without an initializer (§ 6.1.1, p. 205)
- When we explicitly request value initialization by writing an expression of the form `T()` where `T` is the name of a type (The `vector` constructor that takes a single argument to specify the `vector`'s size (§ 3.3.1, p. 98) uses an argument of this kind to value initialize its element initializer.)

Classes must have a default constructor in order to be used in these contexts. Most of these contexts should be fairly obvious.

What may be less obvious is the impact on classes that have data members that do not have a default constructor:

```
class NoDefault {
public:
    NoDefault(const std::string&);  
    // additional members follow, but no other constructors
};  
  
struct A { // my_mem is public by default; see § 7.2 (p. 268)
    NoDefault my_mem;
};  
  
A a;           // error: cannot synthesize a constructor for A  
  
struct B {
    B() {} // error: no initializer for b_member
    NoDefault b_member;
};
```

Best Practices

In practice, it is almost always right to provide a default constructor if other constructors are being defined.

## Using the Default Constructor

The following declaration of `obj` compiles without complaint. However, when we try to use `obj`

```
Sales_data obj(); // ok: but defines a function, not an object
if (obj.isbn() == Primer_5th_ed.isbn()) // error: obj is a function
```

the compiler complains that we cannot apply member access notation to a function. The problem is that, although we intended to declare a default-initialized object, `obj` actually declares a function taking no parameters and returning an object of type `Sales_data`.

The correct way to define an object that uses the default constructor for initialization is to leave off the trailing, empty parentheses:

```
// ok: obj is a default-initialized object
Sales_data obj;
```



It is a common mistake among programmers new to C++ to try to declare an object initialized with the default constructor as follows:

```
Sales_data obj(); // oops! declares a function, not an object
Sales_data obj2; // ok: obj2 is an object, not a function
```

## EXERCISES SECTION 7.5.3

**Exercise 7.43:** Assume we have a class named `NoDefault` that has a constructor that takes an `int`, but has no default constructor. Define a class `C` that has a member of type `NoDefault`. Define the default constructor for `C`.

**Exercise 7.44:** Is the following declaration legal? If not, why not?

```
vector<NoDefault> vec(10);
```

**Exercise 7.45:** What if we defined the `vector` in the previous exercise to hold objects of type `C`?

**Exercise 7.46:** Which, if any, of the following statements are untrue? Why?

- A class must provide at least one constructor.
- A default constructor is a constructor with an empty parameter list.
- If there are no meaningful default values for a class, the class should not provide a default constructor.
- If a class does not define a default constructor, the compiler generates one that initializes each data member to the default value of its associated type.



### 7.5.4 Implicit Class-Type Conversions

As we saw in § 4.11 (p. 159), the language defines several automatic conversions among the built-in types. We also noted that classes can define implicit conversions as well. Every constructor that can be called with a single argument defines an implicit conversion to a class type. Such constructors are sometimes referred to as

**converting constructors.** We'll see in § 14.9 (p. 579) how to define conversions *from* a class type to another type.



A constructor that can be called with a single argument defines an implicit conversion from the constructor's parameter type to the class type.

The `Sales_data` constructors that take a `string` and that take an `istream` both define implicit conversions from those types to `Sales_data`. That is, we can use a `string` or an `istream` where an object of type `Sales_data` is expected:

```
string null_book = "9-999-99999-9";
// constructs a temporary Sales_data object
// with units_sold and revenue equal to 0 and bookNo equal to null_book
item.combine(null_book);
```

Here we call the `Sales_data` `combine` member function with a `string` argument. This call is perfectly legal; the compiler automatically creates a `Sales_data` object from the given `string`. That newly generated (temporary) `Sales_data` is passed to `combine`. Because `combine`'s parameter is a reference to `const`, we can pass a temporary to that parameter.

## Only One Class-Type Conversion Is Allowed

In § 4.11.2 (p. 162) we noted that the compiler will automatically apply only one class-type conversion. For example, the following code is in error because it implicitly uses two conversions:

```
// error: requires two user-defined conversions:
//       (1) convert "9-999-99999-9" to string
//       (2) convert that (temporary) string to Sales_data
item.combine("9-999-99999-9");
```

If we wanted to make this call, we can do so by explicitly converting the character string to either a `string` or a `Sales_data` object:

```
// ok: explicit conversion to string, implicit conversion to Sales_data
item.combine(string("9-999-99999-9"));
// ok: implicit conversion to string, explicit conversion to Sales_data
item.combine(Sales_data("9-999-99999-9"));
```

## Class-Type Conversions Are Not Always Useful

Whether the conversion of a `string` to `Sales_data` is desired depends on how we think our users will use the conversion. In this case, it might be okay. The `string` in `null_book` probably represents a nonexistent ISBN.

More problematic is the conversion from `istream` to `Sales_data`:

```
// uses the istream constructor to build an object to pass to combine
item.combine(cin);
```

This code implicitly converts `cin` to `Sales_data`. This conversion executes the `Sales_data` constructor that takes an `istream`. That constructor creates a (temporary) `Sales_data` object by reading the standard input. That object is then passed to `combine`.

This `Sales_data` object is a temporary (§ 2.4.1, p. 62). We have no access to it once `combine` finishes. Effectively, we have constructed an object that is discarded after we add its value into `item`.

## Suppressing Implicit Conversions Defined by Constructors

We can prevent the use of a constructor in a context that requires an implicit conversion by declaring the constructor as `explicit`:

```
class Sales_data {
public:
    Sales_data() = default;
    Sales_data(const std::string &s, unsigned n, double p):
        bookNo(s), units_sold(n), revenue(p*n) { }
    explicit Sales_data(const std::string &s): bookNo(s) { }
    explicit Sales_data(std::istream&); // remaining members as before
};
```

Now, neither constructor can be used to implicitly create a `Sales_data` object. Neither of our previous uses will compile:

```
item.combine(null_book); // error: string constructor is explicit
item.combine(cin); // error: istream constructor is explicit
```

The `explicit` keyword is meaningful only on constructors that can be called with a single argument. Constructors that require more arguments are not used to perform an implicit conversion, so there is no need to designate such constructors as `explicit`. The `explicit` keyword is used only on the constructor declaration inside the class. It is not repeated on a definition made outside the class body:

```
// error: explicit allowed only on a constructor declaration in a class header
explicit Sales_data::Sales_data(istream& is)
{
    read(is, *this);
}
```

## `explicit` Constructors Can Be Used Only for Direct Initialization

One context in which implicit conversions happen is when we use the copy form of initialization (with an `=`) (§ 3.2.1, p. 84). We cannot use an `explicit` constructor with this form of initialization; we must use direct initialization:

```
Sales_data item1(null_book); // ok: direct initialization
// error: cannot use the copy form of initialization with an explicit constructor
Sales_data item2 = null_book;
```



When a constructor is declared `explicit`, it can be used only with the direct form of initialization (§ 3.2.1, p. 84). Moreover, the compiler will *not* use this constructor in an automatic conversion.

## Explicitly Using Constructors for Conversions

Although the compiler will not use an explicit constructor for an implicit conversion, we can use such constructors explicitly to force a conversion:

```
// ok: the argument is an explicitly constructed Sales_data object
item.combine(Sales_data(null_book));
// ok: static_cast can use an explicit constructor
item.combine(static_cast<Sales_data>(cin));
```

In the first call, we use the `Sales_data` constructor directly. This call constructs a temporary `Sales_data` object using the `Sales_data` constructor that takes a string. In the second call, we use a `static_cast` (§ 4.11.3, p. 163) to perform an explicit, rather than an implicit, conversion. In this call, the `static_cast` uses the `istream` constructor to construct a temporary `Sales_data` object.

## Library Classes with **explicit** Constructors

Some of the library classes that we've used have single-parameter constructors:

- The `string` constructor that takes a single parameter of type `const char*` (§ 3.2.1, p. 84) is not explicit.
- The `vector` constructor that takes a size (§ 3.3.1, p. 98) is explicit.

### EXERCISES SECTION 7.5.4

**Exercise 7.47:** Explain whether the `Sales_data` constructor that takes a `string` should be `explicit`. What are the benefits of making the constructor `explicit`? What are the drawbacks?

**Exercise 7.48:** Assuming the `Sales_data` constructors are not `explicit`, what operations happen during the following definitions

```
string null_isbn("9-999-99999-9");
Sales_data item1(null_isbn);
Sales_data item2("9-999-99999-9");
```

What happens if the `Sales_data` constructors are `explicit`?

**Exercise 7.49:** For each of the three following declarations of `combine`, explain what happens if we call `i.combine(s)`, where `i` is a `Sales_data` and `s` is a `string`:

- (a) `Sales_data &combine(Sales_data);`
- (b) `Sales_data &combine(Sales_data&);`
- (c) `Sales_data &combine(const Sales_data&); const;`

**Exercise 7.50:** Determine whether any of your `Person` class constructors should be `explicit`.

**Exercise 7.51:** Why do you think `vector` defines its single-argument constructor as `explicit`, but `string` does not?



## 7.5.5 Aggregate Classes

An **aggregate class** gives users direct access to its members and has special initialization syntax. A class is an aggregate if

- All of its data members are `public`
- It does not define any constructors
- It has no in-class initializers (§ 2.6.1, p. 73)
- It has no base classes or virtual functions, which are class-related features that we'll cover in Chapter 15

For example, the following class is an aggregate:

```
struct Data {
    int ival;
    string s;
};
```

We can initialize the data members of an aggregate class by providing a braced list of member initializers:

```
// val1.ival = 0; val1.s = string( "Anna" )
Data val1 = { 0, "Anna" };
```

The initializers must appear in declaration order of the data members. That is, the initializer for the first member is first, for the second is next, and so on. The following, for example, is an error:

```
// error: can't use "Anna" to initialize ival, or 1024 to initialize s
Data val2 = { "Anna" , 1024 };
```

As with initialization of array elements (§ 3.5.1, p. 114), if the list of initializers has fewer elements than the class has members, the trailing members are value initialized (§ 3.5.1, p. 114). The list of initializers must not contain more elements than the class has members.

It is worth noting that there are three significant drawbacks to explicitly initializing the members of an object of class type:

- It requires that all the data members of the class be `public`.
- It puts the burden on the user of the class (rather than on the class author) to correctly initialize every member of every object. Such initialization is tedious and error-prone because it is easy to forget an initializer or to supply an inappropriate initializer.
- If a member is added or removed, all initializations have to be updated.

**EXERCISES SECTION 7.5.5**

**Exercise 7.52:** Using our first version of `Sales_data` from § 2.6.1 (p. 72), explain the following initialization. Identify and fix any problems.

```
Sales_data item = {"978-0590353403", 25, 15.99};
```



## 7.5.6 Literal Classes

In § 6.5.2 (p. 239) we noted that the parameters and return type of a `constexpr` function must be literal types. In addition to the arithmetic types, references, and pointers, certain classes are also literal types. Unlike other classes, classes that are literal types may have function members that are `constexpr`. Such members must meet all the requirements of a `constexpr` function. These member functions are implicitly `const` (§ 7.1.2, p. 258).

An aggregate class (§ 7.5.5, p. 298) whose data members are all of literal type is a literal class. A nonaggregate class, that meets the following restrictions, is also a literal class:

- The data members all must have literal type.
- The class must have at least one `constexpr` constructor.
- If a data member has an in-class initializer, the initializer for a member of built-in type must be a constant expression (§ 2.4.4, p. 65), or if the member has class type, the initializer must use the member's own `constexpr` constructor.
- The class must use default definition for its destructor, which is the member that destroys objects of the class type (§ 7.1.5, p. 267).

### `constexpr` Constructors

Although constructors can't be `const` (§ 7.1.4, p. 262), constructors in a literal class can be `constexpr` (§ 6.5.2, p. 239) functions. Indeed, a literal class must provide at least one `constexpr` constructor.

A `constexpr` constructor can be declared as `= default` (§ 7.1.4, p. 264) (or as a deleted function, which we cover in § 13.1.6 (p. 507)). Otherwise, a `constexpr` constructor must meet the requirements of a constructor—meaning it can have no return statement—and of a `constexpr` function—meaning the only executable statement it can have is a `return` statement (§ 6.5.2, p. 239). As a result, the body of a `constexpr` constructor is typically empty. We define a `constexpr` constructor by preceding its declaration with the keyword `constexpr`:

```
class Debug {
public:
    constexpr Debug(bool b = true): hw(b), io(b), other(b) { }
    constexpr Debug(bool h, bool i, bool o):
        hw(h), io(i), other(o) { }
```

C++  
11

```

        constexpr bool any() { return hw || io || other; }
        void set_io(bool b) { io = b; }
        void set_hw(bool b) { hw = b; }
        void set_other(bool b) { hw = b; }
    private:
        bool hw;      // hardware errors other than IO errors
        bool io;      // IO errors
        bool other;   // other errors
    };

```

A `constexpr` constructor must initialize every data member. The initializers must either use a `constexpr` constructor or be a constant expression.

A `constexpr` constructor is used to generate objects that are `constexpr` and for parameters or return types in `constexpr` functions:

```

constexpr Debug io_sub(false, true, false); // debugging IO
if (io_sub.any()) // equivalent to if(true)
    cerr << "print appropriate error messages" << endl;
constexpr Debug prod(false); // no debugging during production
if (prod.any()) // equivalent to if(false)
    cerr << "print an error message" << endl;

```

## EXERCISES SECTION 7.5.6

**Exercise 7.53:** Define your own version of `Debug`.

**Exercise 7.54:** Should the members of `Debug` that begin with `set_` be declared as `constexpr`? If not, why not?

**Exercise 7.55:** Is the `Data` class from § 7.5.5 (p. 298) a literal class? If not, why not? If so, explain why it is literal.

## 7.6 static Class Members

Classes sometimes need members that are associated with the class, rather than with individual objects of the class type. For example, a bank account class might need a data member to represent the current prime interest rate. In this case, we'd want to associate the rate with the class, not with each individual object. From an efficiency standpoint, there'd be no reason for each object to store the rate. Much more importantly, if the rate changes, we'd want each object to use the new value.

### Declaring static Members

We say a member is associated with the class by adding the keyword `static` to its declaration. Like any other member, `static` members can be `public` or `private`. The type of a `static` data member can be `const`, `reference`, `array`, `class` type, and so forth.

As an example, we'll define a class to represent an account record at a bank:

```
class Account {  
public:  
    void calculate() { amount += amount * interestRate; }  
    static double rate() { return interestRate; }  
    static void rate(double);  
private:  
    std::string owner;  
    double amount;  
    static double interestRate;  
    static double initRate();  
};
```

The `static` members of a class exist outside any object. Objects do not contain data associated with `static` data members. Thus, each `Account` object will contain two data members—`owner` and `amount`. There is only one `interestRate` object that will be shared by all the `Account` objects.

Similarly, `static` member functions are not bound to any object; they do not have a `this` pointer. As a result, `static` member functions may not be declared as `const`, and we may not refer to `this` in the body of a `static` member. This restriction applies both to explicit uses of `this` and to implicit uses of `this` by calling a nonstatic member.

## Using a Class `static` Member

We can access a `static` member directly through the scope operator:

```
double r;  
r = Account::rate(); // access a static member using the scope operator
```

Even though `static` members are not part of the objects of its class, we can use an object, reference, or pointer of the class type to access a `static` member:

```
Account ac1;  
Account *ac2 = &ac1;  
// equivalent ways to call the static member rate function  
r = ac1.rate(); // through an Account object or reference  
r = ac2->rate(); // through a pointer to an Account object
```

Member functions can use `static` members directly, without the scope operator:

```
class Account {  
public:  
    void calculate() { amount += amount * interestRate; }  
private:  
    static double interestRate;  
    // remaining members as before  
};
```

## Defining static Members

As with any other member function, we can define a `static` member function inside or outside of the class body. When we define a `static` member outside the class, we do not repeat the `static` keyword. The keyword appears only with the declaration inside the class body:

```
void Account::rate(double newRate)
{
    interestRate = newRate;
}
```



As with any class member, when we refer to a class `static` member outside the class body, we must specify the class in which the member is defined. The `static` keyword, however, is used *only* on the declaration inside the class body.

Because `static` data members are not part of individual objects of the class type, they are not defined when we create objects of the class. As a result, they are not initialized by the class' constructors. Moreover, in general, we may not initialize a `static` member inside the class. Instead, we must define and initialize each `static` data member outside the class body. Like any other object, a `static` data member may be defined only once.

Like global objects (§ 6.1.1, p. 204), `static` data members are defined outside any function. Hence, once they are defined, they continue to exist until the program completes.

We define a `static` data member similarly to how we define class member functions outside the class. We name the object's type, followed by the name of the class, the scope operator, and the member's own name:

```
// define and initialize a static class member
double Account::interestRate = initRate();
```

This statement defines the object named `interestRate` that is a `static` member of class `Account` and has type `double`. Once the class name is seen, the remainder of the definition is in the scope of the class. As a result, we can use `initRate` without qualification as the initializer for `interestRate`. Note also that although `initRate` is `private`, we can use it to initialize `interestRate`. As with any other member definition, a `static` data member definition may access the `private` members of its class.



The best way to ensure that the object is defined exactly once is to put the definition of `static` data members in the same file that contains the definitions of the class noninline member functions.

## In-Class Initialization of static Data Members

Ordinarily, class `static` members may not be initialized in the class body. However, we can provide in-class initializers for `static` members that have `const` integral type and must do so for `static` members that are `constexprs` of literal

type (§ 7.5.6, p. 299). The initializers must be constant expressions. Such members are themselves constant expressions; they can be used where a constant expression is required. For example, we can use an initialized `static` data member to specify the dimension of an array member:

```
class Account {
public:
    static double rate() { return interestRate; }
    static void rate(double);
private:
    static constexpr int period = 30; // period is a constant expression
    double daily_tbl[period];
};
```

If the member is used only in contexts where the compiler can substitute the member's value, then an initialized `const` or `constexpr` `static` need not be separately defined. However, if we use the member in a context in which the value cannot be substituted, then there must be a definition for that member.

For example, if the only use we make of `period` is to define the dimension of `daily_tbl`, there is no need to define `period` outside of `Account`. However, if we omit the definition, it is possible that even seemingly trivial changes to the program might cause the program to fail to compile because of the missing definition. For example, if we pass `Account::period` to a function that takes a `const int&`, then `period` must be defined.

If an initializer is provided inside the class, the member's definition must not specify an initial value:

```
// definition of a static member with no initializer
constexpr int Account::period; // initializer provided in the class definition
```



Even if a `const static` data member is initialized in the class body, that member ordinarily should be defined outside the class definition.

## static Members Can Be Used in Ways Ordinary Members Can't

As we've seen, `static` members exist independently of any other object. As a result, they can be used in ways that would be illegal for nonstatic data members. As one example, a `static` data member can have incomplete type (§ 7.3.3, p. 278). In particular, a `static` data member can have the same type as the class type of which it is a member. A nonstatic data member is restricted to being declared as a pointer or a reference to an object of its class:

```
class Bar {
public:
    // ...
private:
    static Bar mem1; // ok: static member can have incomplete type
    Bar *mem2;       // ok: pointer member can have incomplete type
    Bar mem3;        // error: data members must have complete type
};
```

Another difference between `static` and ordinary members is that we can use a `static` member as a default argument (§ 6.5.1, p. 236):

```
class Screen {  
public:  
    // bkground refers to the static member  
    // declared later in the class definition  
    Screen& clear(char = bkground);  
private:  
    static const char bkground;  
};
```

A nonstatic data member may not be used as a default argument because its value is part of the object of which it is a member. Using a nonstatic data member as a default argument provides no object from which to obtain the member's value and so is an error.

## EXERCISES SECTION 7.6

**Exercise 7.56:** What is a `static` class member? What are the advantages of `static` members? How do they differ from ordinary members?

**Exercise 7.57:** Write your own version of the `Account` class.

**Exercise 7.58:** 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;
```

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

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



## 8.1 The IO Classes

The IO types and objects that we've used so far manipulate `char` data. By default these objects are connected to the user's console window. Of course, real programs cannot be limited to doing IO solely to or from a console window. Programs often need to read or write named files. Moreover, it can be convenient to use IO operations to process the characters in a `string`. Applications also may have to read and write languages that require wide-character support.

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

Table 8.1: IO Library Types and Headers

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

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

## Relationships among the IO Types

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

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

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

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



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

### 8.1.1 No Copy or Assign for IO Objects



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

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

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

### 8.1.2 Condition States

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

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

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

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

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

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

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

### Interrogating the State of a Stream

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

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

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

The library also defines a set of functions to interrogate the state of these flags. The `good` operation returns `true` if none of the error bits is set. The `bad`, `fail`,

**Table 8.2: IO Library Condition State**

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

and `eof` operations return `true` when the corresponding bit is on. In addition, `fail` returns `true` if `bad` is set. By implication, the right way to determine the overall state of a stream is to use either `good` or `fail`. Indeed, the code that is executed when we use a stream as a condition is equivalent to calling `!fail()`. The `eof` and `bad` operations reveal only whether those specific errors have occurred.

## Managing the Condition State

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

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

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

The version of `clear` that takes an argument expects an `iostate` value that represents the new state of the stream. To turn off a single condition, we use the `rdstate` member and the bitwise operators to produce the desired new state.

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

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

### EXERCISES SECTION 8.1.2

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

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

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

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

### 8.1.3 Managing the Output Buffer

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

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

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

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

- The program completes normally. All output buffers are flushed as part of the `return` from `main`.
- At some indeterminate time, the buffer can become full, in which case it will be flushed before writing the next value.
- We can flush the buffer explicitly using a manipulator such as `endl` (§ 1.2, p. 7).
- We can use the `unitbuf` manipulator to set the stream's internal state to empty the buffer after each output operation. By default, `unitbuf` is set for `cerr`, so that writes to `cerr` are flushed immediately.
- An output stream might be tied to another stream. In this case, the output stream is flushed whenever the stream to which it is tied is read or written. By default, `cin` and `cerr` are both tied to `cout`. Hence, reading `cin` or writing to `cerr` flushes the buffer in `cout`.

## Flushing the Output Buffer

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

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

## The `unitbuf` Manipulator

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

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

### CAUTION: BUFFERS ARE NOT FLUSHED IF THE PROGRAM CRASHES

Output buffers are *not* flushed if the program terminates abnormally. When a program crashes, it is likely that data the program wrote may be sitting in an output buffer waiting to be printed.

When you debug a program that has crashed, it is essential to make sure that any output you *think* should have been written was actually flushed. Countless hours of programmer time have been wasted tracking through code that appeared not to have executed when in fact the buffer had not been flushed and the output was pending when the program crashed.

## Tying Input and Output Streams Together

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

```
cin >> ival;
```

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



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

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

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

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

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

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



## 8.2 File Input and Output

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

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

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

**Table 8.3: `fstream`-Specific Operations**

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

## 8.2.1 Using File Stream Objects



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

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

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

This code defines `in` as an input stream that is initialized to read from the file named by the string argument `ifile`. It defines `out` as an output stream that is not yet associated with a file. With the new standard, file names can be either library strings or C-style character arrays (§ 3.5.4, p. 122). Previous versions of the library allowed only C-style character arrays.

C++ 11

### Using an `fstream` in Place of an `iostream`&

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

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

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

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

## The `open` and `close` Members

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

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

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

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

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

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

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

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

## Automatic Construction and Destruction

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

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

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

Because `input` is defined inside the block that forms the `for` body, it is created and destroyed on each iteration (§ 6.1.1, p. 205). When an `fstream` object goes out of scope, the file it is bound to is automatically closed. On the next iteration, `input` is created anew.

*Note*

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

**EXERCISES SECTION 8.2.1**

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

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

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

## 8.2.2 File Modes



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

**Table 8.4: File Modes**

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

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

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

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

## Opening a File in `out` Mode Discards Existing Data

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

```
// file1 is truncated in each of these cases
ofstream out("file1");      // out and trunc are implicit
ofstream out2("file1", ofstream::out);    // trunc is implicit
ofstream out3("file1", ofstream::out | ofstream::trunc);
// to preserve the file's contents, we must explicitly specify app mode
ofstream app("file2", ofstream::app);     // out is implicit
ofstream app2("file2", ofstream::out | ofstream::app);
```



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

## File Mode Is Determined Each Time `open` Is Called

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

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

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



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

### EXERCISES SECTION 8.2.2

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

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

## 8.3 `string` Streams

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

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

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

**Table 8.5: `stringstream`-Specific Operations**

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

### 8.3.1 Using an `istringstream`

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

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

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

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

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

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

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

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

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

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

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

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

## EXERCISES SECTION 8.3.1

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

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

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

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

### 8.3.2 Using `ostringstream`

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

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

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

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

#### EXERCISES SECTION 8.3.2

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

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

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

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



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



## 9.1 Overview of the Sequential Containers

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

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

**Table 9.1: Sequential Container Types**

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

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

For example, `string` and `vector` hold their elements in contiguous memory. Because elements are contiguous, it is fast to compute the address of an element

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

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

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

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

C++  
11

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

## Deciding Which Sequential Container to Use



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

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

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

- If the program needs to insert elements in the middle of the container only while reading input, and subsequently needs random access to the elements:
  - First, decide whether you actually need to add elements in the middle of a container. It is often easier to append to a `vector` and then call the library `sort` function (which we shall cover in § 10.2.3 (p. 384)) to reorder the container when you’re done with input.
  - If you must insert into the middle, consider using a `list` for the input phase. Once the input is complete, copy the `list` into a `vector`.

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



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

## EXERCISES SECTION 9.1

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

- (a) Read a fixed number of words, inserting them in the container alphabetically as they are entered. We’ll see in the next chapter that associative containers are better suited to this problem.
- (b) Read an unknown number of words. Always insert new words at the back. Remove the next value from the front.
- (c) Read an unknown number of integers from a file. Sort the numbers and then print them to standard output.



## 9.2 Container Library Overview

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

- Some operations (Table 9.2 (p. 330)) are provided by all container types.
- Other operations are specific to the sequential (Table 9.3 (p. 335)), the associative (Table 11.7 (p. 438)), or the unordered (Table 11.8 (p. 445)) containers.
- Still others are common to only a smaller subset of the containers.

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

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

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

## Constraints on Types That a Container Can Hold

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

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

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

C++  
11



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

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

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

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

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

## EXERCISES SECTION 9.2

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

**Table 9.2: Container Operations**

<b>Type Aliases</b>	
<code>iterator</code>	Type of the iterator for this container type
<code>const_iterator</code>	Iterator type that can read but not change its elements
<code>size_type</code>	Unsigned integral type big enough to hold the size of the largest possible container of this container type
<code>difference_type</code>	Signed integral type big enough to hold the distance between two iterators
<code>value_type</code>	Element type
<code>reference</code>	Element's lvalue type; synonym for <code>value_type&amp;</code>
<code>const_reference</code>	Element's const lvalue type (i.e., <code>const value_type&amp;</code> )
<b>Construction</b>	
<code>C c;</code>	Default constructor, empty container ( <code>array</code> ; see p. 336)
<code>C c1(c2);</code>	Construct <code>c1</code> as a copy of <code>c2</code>
<code>C c(b, e);</code>	Copy elements from the range denoted by iterators <code>b</code> and <code>e</code> ; <b>(not valid for <code>array</code>)</b>
<code>C c{a,b,c...};</code>	List initialize <code>c</code>
<b>Assignment and swap</b>	
<code>c1 = c2</code>	Replace elements in <code>c1</code> with those in <code>c2</code>
<code>c1 = {a,b,c...}</code>	Replace elements in <code>c1</code> with those in the list <b>(not valid for <code>array</code>)</b>
<code>a.swap(b)</code>	Swap elements in <code>a</code> with those in <code>b</code>
<code>swap(a, b)</code>	Equivalent to <code>a.swap(b)</code>
<b>Size</b>	
<code>c.size()</code>	Number of elements in <code>c</code> <b>(not valid for <code>forward_list</code>)</b>
<code>c.max_size()</code>	Maximum number of elements <code>c</code> can hold
<code>c.empty()</code>	false if <code>c</code> has any elements, true otherwise
<b>Add/Remove Elements (not valid for <code>array</code>)</b>	
<i>Note: the interface to these operations varies by container type</i>	
<code>c.insert(args)</code>	Copy element(s) as specified by <code>args</code> into <code>c</code>
<code>c.emplace(inits)</code>	Use <code>inits</code> to construct an element in <code>c</code>
<code>c.erase(args)</code>	Remove element(s) specified by <code>args</code>
<code>c.clear()</code>	Remove all elements from <code>c</code> ; returns void
<b>Equality and Relational Operators</b>	
<code>==, !=</code>	Equality valid for all container types
<code>&lt;, &lt;=, &gt;, &gt;=</code>	Relational operators <b>(not valid for unordered associative containers)</b>
<b>Obtain Iterators</b>	
<code>c.begin(), c.end()</code>	Return iterator to the first, one past the last element in <code>c</code>
<code>c.cbegin(), c.cend()</code>	Return <code>const_iterator</code>
<b>Additional Members of Reversible Containers (not valid for <code>forward_list</code>)</b>	
<code>reverse_iterator</code>	Iterator that addresses elements in reverse order
<code>const_reverse_iterator</code>	Reverse iterator that cannot write the elements
<code>c.rbegin(), c.rend()</code>	Return iterator to the last, one past the first element in <code>c</code>
<code>c.crbegin(), c.crend()</code>	Return <code>const_reverse_iterator</code>



## 9.2.1 Iterators

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

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

### Iterator Ranges



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

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

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

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

`[ begin, end )`

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

#### REQUIREMENTS ON ITERATORS FORMING AN ITERATOR RANGE

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

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



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

## Programming Implications of Using Left-Inclusive Ranges

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

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

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

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

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

### EXERCISES SECTION 9.2.1

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

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

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

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

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

### 9.2.2 Container Type Members

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

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

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

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

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

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

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

### EXERCISES SECTION 9.2.2

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

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

### 9.2.3 begin and end Members



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

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

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

The functions that do not begin with a `c` are overloaded. That is, there are actually two members named `begin`. One is a `const` member (§ 7.1.2, p. 258) that returns the container's `const_iterator` type. The other is nonconst and returns the container's `iterator` type. Similarly for `rbegin`, `end`, and `rend`. When we

call one of these members on a `nonconst` object, we get the version that returns `iterator`. We get a `const` version of the iterators *only* when we call these functions on a `const` object. As with pointers and references to `const`, we can convert a plain iterator to the corresponding `const_iterator`, but not vice versa.

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

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

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



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

## EXERCISES SECTION 9.2.3

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

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

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



### 9.2.4 Defining and Initializing a Container

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

#### Initializing a Container as a Copy of Another Container

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

To create a container as a copy of another container, the container and element types must match. When we pass iterators, there is no requirement that the container types be identical. Moreover, the element types in the new and original

**Table 9.3: Defining and Initializing Containers**

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

containers can differ as long as it is possible to convert (§ 4.11, p. 159) the elements we're copying to the element type of the container we are initializing:

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



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

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

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

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

## List Initialization

C++ 11

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

```
// each container has three elements, initialized from the given initializers
list<string> authors = {"Milton", "Shakespeare", "Austen"};
vector<const char*> articles = {"a", "an", "the"};
```

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

## Sequential Container Size-Related Constructors

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

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

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



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

## Library arrays Have Fixed Size

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

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

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

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

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

The fixed-size nature of `arrays` also affects the behavior of the constructors that `array` does define. Unlike the other containers, a default-constructed `array`

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

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

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

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

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

### EXERCISES SECTION 9.2.4

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

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

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

## 9.2.5 Assignment and swap

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

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

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

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

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

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

**Table 9.4: Container Assignment Operations**

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



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

## Using `assign` (Sequential Containers Only)

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

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

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



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

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

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

## Using `swap`

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

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

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



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

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

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

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

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

C++  
11

## EXERCISES SECTION 9.2.5

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



### 9.2.6 Container Size Operations

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

### 9.2.7 Relational Operators

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

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

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

The following examples illustrate how these operators work:

```

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

```

## Relational Operators Use Their Element's Relational Operator



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

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

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

### EXERCISES SECTION 9.2.7

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

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

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

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

## 9.3 Sequential Container Operations

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

### 9.3.1 Adding Elements to a Sequential Container

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



When we use these operations, we must remember that the containers use different strategies for allocating elements and that these strategies affect performance. Adding elements anywhere but at the end of a `vector` or `string`, or anywhere but the beginning or end of a `deque`, requires elements to be moved.

Moreover, adding elements to a vector or a string may cause the entire object to be reallocated. Reallocating an object requires allocating new memory and moving elements from the old space to the new.

## Using `push_back`

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

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

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

The call to `push_back` creates a new element at the end of `container`, increasing the size of `container` by 1. The value of that element is a copy of `word`. The type of `container` can be any of `list`, `vector`, or `deque`.

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

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

### KEY CONCEPT: CONTAINER ELEMENTS ARE COPIES

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

## Using `push_front`

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

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

This loop adds the elements 0, 1, 2, 3 to the beginning of `ilist`. Each element is inserted at the *new beginning* of the list. That is, when we insert 1, it goes in

front of 0, and 2 in front of 1, and so forth. Thus, the elements added in a loop such as this one wind up in reverse order. After executing this loop, `iList` holds the sequence 3, 2, 1, 0.

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

**Table 9.5: Operations That Add Elements to a Sequential Container**

These operations change the size of the container; they are not supported by `array`.

`forward_list` has special versions of `insert` and `emplace`; see § 9.3.4 (p. 350).

`push_back` and `emplace_back` not valid for `forward_list`.

`push_front` and `emplace_front` not valid for `vector` or `string`.

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



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

## Adding Elements at a Specified Point in the Container

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

Each of the `insert` functions takes an iterator as its first argument. The iterator indicates where in the container to put the element(s). It can refer to any position in the container, including one past the end of the container. Because the iterator

might refer to a nonexistent element off the end of the container, and because it is useful to have a way to insert elements at the beginning of a container, element(s) are inserted *before* the position denoted by the iterator. For example, this statement

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

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

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

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



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

## Inserting a Range of Elements

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

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

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

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

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

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

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

## Using the Return from `insert`

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

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



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

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

## Using the Emplace Operations

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

C++ 11

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

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

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

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

```
// iter refers to an element in c, which holds Sales_data elements
c.emplace_back(); // uses the Sales_data default constructor
c.emplace(iter, "999-99999999"); // uses Sales_data(string)
// uses the Sales_data constructor that takes an ISBN, a count, and a price
c.emplace_front("978-0590353403", 25, 15.99);
```



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

## EXERCISES SECTION 9.3.1

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

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

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

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

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

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



### 9.3.2 Accessing Elements

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

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

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

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

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

**Table 9.6: Operations to Access Elements in a Sequential Container**

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

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



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

## The Access Members Return References

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

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

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

## Subscripting and Safe Random Access

The containers that provide fast random access (`string`, `vector`, `deque`, and `array`) also provide the subscript operator (§ 3.3.3, p. 102). As we've seen, the

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

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

Rather than define each of these operations as members of each container type, the standard library defines a set of **generic algorithms**: “algorithms” because they implement common classical algorithms such as sorting and searching, and “generic” because they operate on elements of differing type and across multiple container types—not only library types such as `vector` or `list`, but also the built-in array type—and, as we shall see, over other kinds of sequences as well.



## 10.1 Overview

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

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

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

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

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

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

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

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

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

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

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

## How the Algorithms Work

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

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

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

## Iterators Make the Algorithms Container Independent, ...

All but the second step in the `find` function can be handled by iterator operations: The iterator dereference operator gives access to an element's value; if a matching element is found, `find` can return an iterator to that element; the iterator increment operator moves to the next element; the “off-the-end” iterator will indicate when `find` has reached the end of its given sequence; and `find` can return the off-the-end iterator (§ 9.2.1, p. 331) to indicate that the given value wasn't found.

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

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

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

### EXERCISES SECTION 10.1

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

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

#### KEY CONCEPT: ALGORITHMS NEVER EXECUTE CONTAINER OPERATIONS

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

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



## 10.2 A First Look at the Algorithms

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

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

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

### 10.2.1 Read-Only Algorithms



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

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

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

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



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

## Algorithms and Element Types

The fact that `accumulate` uses its third argument as the starting point for the summation has an important implication: It must be possible to add the element type to the type of the sum. That is, the elements in the sequence must match or be convertible to the type of the third argument. In this example, the elements in `vec` might be `ints`, or they might be `double`, or `long long`, or any other type that can be added to an `int`.

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

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

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

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

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



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



## Algorithms That Operate on Two Sequences

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

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

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

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



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

### EXERCISES SECTION 10.2.1

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

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

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



## 10.2.2 Algorithms That Write Container Elements

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

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

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

### KEY CONCEPT: ITERATOR ARGUMENTS

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

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

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

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

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

### Algorithms Do Not Check Write Operations



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

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

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

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

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

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

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

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



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

## Introducing `back_inserter`

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

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

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

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

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

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

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

## Copy Algorithms

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

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

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

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

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

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

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

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

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

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

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

### 10.2.3 Algorithms That Reorder Container Elements



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

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

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

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

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

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

## EXERCISES SECTION 10.2.2

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

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

- (a) 

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

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

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

## Eliminating Duplicates

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

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

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

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

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



## Using `unique`

Once `words` is sorted, we want to keep only one copy of each word. The `unique` algorithm rearranges the input range to “eliminate” adjacent duplicated entries,

and returns an iterator that denotes the end of the range of the unique values. After the call to `unique`, the vector holds

fox	jumps	over	quick	red	slow	the	turtle	???	???
-----	-------	------	-------	-----	------	-----	--------	-----	-----



`end_unique`  
(one past the last unique element)

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



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

## Using Container Operations to Remove Elements



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

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

### EXERCISES SECTION 10.2.3

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

**Exercise 10.10:** Why do you think the algorithms don’t change the size of containers?

## 10.3 Customizing Operations

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

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



### 10.3.1 Passing a Function to an Algorithm

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

#### Predicates

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

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

```
// comparison function to be used to sort by word length
bool isShorter(const string &s1, const string &s2)
{
    return s1.size() < s2.size();
}
// sort on word length, shortest to longest
sort(words.begin(), words.end(), isShorter);
```

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

#### Sorting Algorithms

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

Ordinarily, we don't care about the relative order of equal elements in a sorted sequence. After all, they're equal. However, in this case, we have defined "equal"

to mean “have the same length.” Elements that have the same length still differ from one another when we view their contents. By calling `stable_sort`, we can maintain alphabetical order among those elements that have the same length:

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

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

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

### EXERCISES SECTION 10.3.1

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

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

**Exercise 10.13:** The library defines an algorithm named `partition` that takes a predicate and partitions the container so that values for which the predicate is `true` appear in the first part and those for which the predicate is `false` appear in the second part. The algorithm returns an iterator just past the last element for which the predicate returned `true`. Write a function that takes a `string` and returns a `bool` indicating whether the `string` has five characters or more. Use that function to partition `words`. Print the elements that have five or more characters.

### 10.3.2 Lambda Expressions

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

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

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

```

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

```

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

We can use the library `find_if` algorithm to find an element that has a particular size. Like `find` (§ 10.1, p. 376), the `find_if` algorithm takes a pair of iterators denoting a range. Unlike `find`, the third argument to `find_if` is a predicate. The `find_if` algorithm calls the given predicate on each element in the input range. It returns the first element for which the predicate returns a nonzero value, or its end iterator if no such element is found.

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

## Introducing Lambdas

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

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

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

$$[\text{capture list}] (\text{parameter list}) \rightarrow \text{return type} \{ \text{function body} \}$$

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

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

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

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

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

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

Omitting the parentheses and the parameter list in a lambda is equivalent to specifying an empty parameter list. Hence, when we call `f`, the argument list is empty. If we omit the return type, the lambda has an inferred return type that depends on the code in the function body. If the function body is just a `return` statement, the return type is inferred from the type of the expression that is returned. Otherwise, the return type is `void`.



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

## Passing Arguments to a Lambda

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

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

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

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

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

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

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

## Using the Capture List

We're now ready to solve our original problem, which is to write a callable expression that we can pass to `find_if`. We want an expression that will compare the

length of each string in the input sequence with the value of the `sz` parameter in the `biggies` function.

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

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

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

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

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

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



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

## Calling `find_if`

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

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

The call to `find_if` returns an iterator to the first element that is at least as long as the given `sz`, or a copy of `words.end()` if no such element exists.

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

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

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

## The `for_each` Algorithm

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

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

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

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



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

## Putting It All Together

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

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

## EXERCISES SECTION 10.3.2

**Exercise 10.14:** Write a lambda that takes two ints and returns their sum.

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

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

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

**Exercise 10.18:** Rewrite `biggies` to use `partition` instead of `find_if`. We described the `partition` algorithm in exercise 10.13 in § 10.3.1 (p. 387).

**Exercise 10.19:** Rewrite the previous exercise to use `stable_partition`, which like `stable_sort` maintains the original element order in the partitioned sequence.

### 10.3.3 Lambda Captures and Returns

When we define a lambda, the compiler generates a new (unnamed) class type that corresponds to that lambda. We'll see how these classes are generated in § 14.8.1 (p. 572). For now, what's useful to understand is that when we pass a lambda to a function, we are defining both a new type and an object of that type: The argument is an unnamed object of this compiler-generated class type. Similarly, when we use `auto` to define a variable initialized by a lambda, we are defining an object of the type generated from that lambda.

By default, the class generated from a lambda contains a data member corresponding to the variables captured by the lambda. Like the data members of any class, the data members of a lambda are initialized when a lambda object is created.

#### Capture by Value

Similar to parameter passing, we can capture variables by value or by reference. Table 10.1 (p. 395) covers the various ways we can form a capture list. So far, our lambdas have captured variables by value. As with a parameter passed by value, it must be possible to copy such variables. Unlike parameters, the value of a captured variable is copied when the lambda is created, not when it is called:

```
void fcn1()
{
    size_t v1 = 42; // local variable
    // copies v1 into the callable object named f
    auto f = [v1] { return v1; };
    v1 = 0;
    auto j = f(); // j is 42; f stored a copy of v1 when we created it
}
```

Because the value is copied when the lambda is created, subsequent changes to a captured variable have no effect on the corresponding value inside the lambda.

## Capture by Reference

We can also define lambdas that capture variables by reference. For example:

```
void fcn2()
{
    size_t v1 = 42; // local variable
    // the object f2 contains a reference to v1
    auto f2 = [&v1] { return v1; };
    v1 = 0;
    auto j = f2(); // j is 0; f2 refers to v1; it doesn't store it
}
```

The & before v1 indicates that v1 should be captured as a reference. A variable captured by reference acts like any other reference. When we use the variable inside the lambda body, we are using the object to which that reference is bound. In this case, when the lambda returns v1, it returns the value of the object to which v1 refers.

Reference captures have the same problems and restrictions as reference returns (§ 6.3.2, p. 225). If we capture a variable by reference, we must be *certain* that the referenced object exists at the time the lambda is executed. The variables captured by a lambda are local variables. These variables cease to exist once the function completes. If it is possible for a lambda to be executed after the function finishes, the local variables to which the capture refers no longer exist.

Reference captures are sometimes necessary. For example, we might want our `biggies` function to take a reference to an `ostream` on which to write and a character to use as the separator:

```
void biggies(vector<string> &words,
             vector<string>::size_type sz,
             ostream &os = cout, char c = ' ')
{
    // code to reorder words as before
    // statement to print count revised to print to os
    for_each(words.begin(), words.end(),
              [&os, c](const string &s) { os << s << c; });
}
```

We cannot copy `ostream` objects (§ 8.1.1, p. 311); the only way to capture `os` is by reference (or through a pointer to `os`).

When we pass a lambda to a function, as in this call to `for_each`, the lambda executes immediately. Capturing `os` by reference is fine, because the variables in `biggies` exist while `for_each` is running.

We can also return a lambda from a function. The function might directly return a callable object or the function might return an object of a class that has a callable object as a data member. If the function returns a lambda, then—for the same reasons that a function must not return a reference to a local variable—that lambda must not contain reference captures.



When we capture a variable by reference, we must ensure that the variable exists at the time that the lambda executes.

### ADVICE: KEEP YOUR LAMBDA CAPTURES SIMPLE

A lambda capture stores information between the time the lambda is created (i.e., when the code that defines the lambda is executed) and the time (or times) the lambda itself is executed. It is the programmer's responsibility to ensure that whatever information is captured has the intended meaning each time the lambda is executed.

Capturing an ordinary variable—an `int`, a `string`, or other nonpointer type—by value is usually straightforward. In this case, we only need to care whether the variable has the value we need when we capture it.

If we capture a pointer or iterator, or capture a variable by reference, we must ensure that the object bound to that iterator, pointer, or reference still exists, whenever the lambda *executes*. Moreover, we need to ensure that the object has the intended value. Code that executes between when the lambda is created and when it executes might change the value of the object to which the lambda capture points (or refers). The value of the object at the time the pointer (or reference) was captured might have been what we wanted. The value of that object when the lambda executes might be quite different.

As a rule, we can avoid potential problems with captures by minimizing the data we capture. Moreover, if possible, avoid capturing pointers or references.

## Implicit Captures

Rather than explicitly listing the variables we want to use from the enclosing function, we can let the compiler infer which variables we use from the code in the lambda's body. To direct the compiler to infer the capture list, we use an `&` or `=` in the capture list. The `&` tells the compiler to capture by reference, and the `=` says the values are captured by value. For example, we can rewrite the lambda that we passed to `find_if` as

```
// sz implicitly captured by value
wc = find_if(words.begin(), words.end(),
             [=](const string &s)
                 { return s.size() >= sz; });
```

If we want to capture some variables by value and others by reference, we can mix implicit and explicit captures:

```
void biggies(vector<string> &words,
             vector<string>::size_type sz,
             ostream &os = cout, char c = ' ')
{
    // other processing as before
    // os implicitly captured by reference; c explicitly captured by value
    for_each(words.begin(), words.end(),
              [&, c](const string &s) { os << s << c; });
    // os explicitly captured by reference; c implicitly captured by value
    for_each(words.begin(), words.end(),
              [=, &os](const string &s) { os << s << c; });
}
```

When we mix implicit and explicit captures, the first item in the capture list must be an & or =. That symbol sets the default capture mode as by reference or by value, respectively.

When we mix implicit and explicit captures, the explicitly captured variables must use the alternate form. That is, if the implicit capture is by reference (using &), then the explicitly named variables must be captured by value; hence their names may not be preceded by an &. Alternatively, if the implicit capture is by value (using =), then the explicitly named variables must be preceded by an & to indicate that they are to be captured by reference.

**Table 10.1: Lambda Capture List**

[ ]	Empty capture list. The lambda may not use variables from the enclosing function. A lambda may use local variables only if it captures them.
[ <i>names</i> ]	<i>names</i> is a comma-separated list of names local to the enclosing function. By default, variables in the capture list are copied. A name preceded by & is captured by reference.
[ & ]	Implicit by reference capture list. Entities from the enclosing function used in the lambda body are used by reference.
[ = ]	Implicit by value capture list. Entities from the enclosing function used in the lambda body are copied into the lambda body.
[ &, <i>identifier_list</i> ]	<i>identifier_list</i> is a comma-separated list of zero or more variables from the enclosing function. These variables are captured by value; any implicitly captured variables are captured by reference. The names in <i>identifier_list</i> must not be preceded by an &.
[ =, <i>reference_list</i> ]	Variables included in the <i>reference_list</i> are captured by reference; any implicitly captured variables are captured by value. The names in <i>reference_list</i> may not include <code>this</code> and must be preceded by an &.

## Mutable Lambdas

By default, a lambda may not change the value of a variable that it copies by value. If we want to be able to change the value of a captured variable, we must follow the parameter list with the keyword `mutable`. Lambdas that are mutable may not omit the parameter list:

```
void fcn3()
{
    size_t v1 = 42; // local variable
    // f can change the value of the variables it captures
    auto f = [v1] () mutable { return ++v1; };
    v1 = 0;
    auto j = f(); // j is 43
}
```

Whether a variable captured by reference can be changed (as usual) depends only on whether that reference refers to a `const` or `nonconst` type:

```

void fcn4()
{
    size_t v1 = 42; // local variable
    // v1 is a reference to a nonconst variable
    // we can change that variable through the reference inside f2
    auto f2 = [&v1] { return ++v1; };
    v1 = 0;
    auto j = f2(); // j is 1
}

```

## Specifying the Lambda Return Type

The lambdas we've written so far contain only a single `return` statement. As a result, we haven't had to specify the return type. By default, if a lambda body contains any statements other than a `return`, that lambda is assumed to return `void`. Like other functions that return `void`, lambdas inferred to return `void` may not return a value.

As a simple example, we might use the library `transform` algorithm and a lambda to replace each negative value in a sequence with its absolute value:

```

transform(vi.begin(), vi.end(), vi.begin(),
         [](int i) { return i < 0 ? -i : i; });

```

The `transform` function takes three iterators and a callable. The first two iterators denote an input sequence and the third is a destination. The algorithm calls the given callable on each element in the input sequence and writes the result to the destination. As in this call, the destination iterator can be the same as the iterator denoting the start of the input. When the input iterator and the destination iterator are the same, `transform` replaces each element in the input range with the result of calling the callable on that element.

In this call, we passed a lambda that returns the absolute value of its parameter. The lambda body is a single `return` statement that returns the result of a conditional expression. We need not specify the return type, because that type can be inferred from the type of the conditional operator.

However, if we write the seemingly equivalent program using an `if` statement, our code won't compile:

```

// error: cannot deduce the return type for the lambda
transform(vi.begin(), vi.end(), vi.begin(),
         [](int i) { if (i < 0) return -i; else return i; });

```

This version of our lambda infers the return type as `void` but we returned a value.

 When we need to define a return type for a lambda, we must use a trailing return type (§ 6.3.3, p. 229):

```

transform(vi.begin(), vi.end(), vi.begin(),
         []()> int
         { if (i < 0) return -i; else return i; });

```

In this case, the fourth argument to `transform` is a lambda with an empty capture list, which takes a single parameter of type `int` and returns a value of type `int`. Its function body is an `if` statement that returns the absolute value of its parameter.

### EXERCISES SECTION 10.3.3

**Exercise 10.20:** The library defines an algorithm named `count_if`. Like `find_if`, this function takes a pair of iterators denoting an input range and a predicate that it applies to each element in the given range. `count_if` returns a count of how often the predicate is true. Use `count_if` to rewrite the portion of our program that counted how many words are greater than length 6.

**Exercise 10.21:** Write a lambda that captures a local `int` variable and decrements that variable until it reaches 0. Once the variable is 0 additional calls should no longer decrement the variable. The lambda should return a `bool` that indicates whether the captured variable is 0.

### 10.3.4 Binding Arguments



Lambda expressions are most useful for simple operations that we do not need to use in more than one or two places. If we need to do the same operation in many places, we should usually define a function rather than writing the same lambda expression multiple times. Similarly, if an operation requires many statements, it is ordinarily better to use a function.

It is usually straightforward to use a function in place of a lambda that has an empty capture list. As we've seen, we can use either a lambda or our `isShorter` function to order the `vector` on word length. Similarly, it would be easy to replace the lambda that printed the contents of our `vector` by writing a function that takes a `string` and prints the given `string` to the standard output.

However, it is not so easy to write a function to replace a lambda that captures local variables. For example, the lambda that we used in the call to `find_if` compared a `string` with a given size. We can easily write a function to do the same work:

```
bool check_size(const string &s, string::size_type sz)
{
    return s.size() >= sz;
}
```

However, we can't use this function as an argument to `find_if`. As we've seen, `find_if` takes a unary predicate, so the callable passed to `find_if` must take a single argument. The lambda that `biggies` passed to `find_if` used its capture list to store `sz`. In order to use `check_size` in place of that lambda, we have to figure out how to pass an argument to the `sz` parameter.

### The Library `bind` Function

We can solve the problem of passing a size argument to `check_size` by using a new library function named `bind`, which is defined in the functional header. The `bind` function can be thought of as a general-purpose function adaptor (§ 9.6, p. 368). It takes a callable object and generates a new callable that "adapts" the parameter list of the original object.

C++  
11

The general form of a call to bind is:

```
auto newCallable = bind(callable, arg_list);
```

where *newCallable* is itself a callable object and *arg\_list* is a comma-separated list of arguments that correspond to the parameters of the given *callable*. That is, when we call *newCallable*, *newCallable* calls *callable*, passing the arguments in *arg\_list*.

The arguments in *arg\_list* may include names of the form *\_n*, where *n* is an integer. These arguments are “placeholders” representing the parameters of *newCallable*. They stand “in place of” the arguments that will be passed to *newCallable*. The number *n* is the position of the parameter in the generated callable: *\_1* is the first parameter in *newCallable*, *\_2* is the second, and so forth.

## Binding the sz Parameter of `check_size`

As a simple example, we’ll use bind to generate an object that calls `check_size` with a fixed value for its size parameter as follows:

```
// check6 is a callable object that takes one argument of type string
// and calls check_size on its given string and the value 6
auto check6 = bind(check_size, _1, 6);
```

This call to bind has only one placeholder, which means that *check6* takes a single argument. The placeholder appears first in *arg\_list*, which means that the parameter in *check6* corresponds to the first parameter of `check_size`. That parameter is a `const string&`, which means that the parameter in *check6* is also a `const string&`. Thus, a call to *check6* must pass an argument of type `string`, which *check6* will pass as the first argument to `check_size`.

The second argument in *arg\_list* (i.e., the third argument to bind) is the value 6. That value is bound to the second parameter of `check_size`. Whenever we call *check6*, it will pass 6 as the second argument to `check_size`:

```
string s = "hello";
bool b1 = check6(s); // check6(s) calls check_size(s, 6)
```

Using bind, we can replace our original lambda-based call to `find_if`

```
auto wc = find_if(words.begin(), words.end(),
                   [sz](const string &a)
```

with a version that uses `check_size`:

```
auto wc = find_if(words.begin(), words.end(),
                   bind(check_size, _1, sz));
```

This call to bind generates a callable object that binds the second argument of `check_size` to the value of *sz*. When `find_if` calls this object on the strings in *words* those calls will in turn call `check_size` passing the given `string` and *sz*. So, `find_if` (effectively) will call `check_size` on each `string` in the input range and compare the size of that `string` to *sz*.

## Using `placeholders` Names

The `_n` names are defined in a namespace named `placeholders`. That namespace is itself defined inside the `std` namespace (§ 3.1, p. 82). To use these names, we must supply the names of both namespaces. As with our other examples, our calls to `bind` assume the existence of appropriate `using` declarations. For example, the `using` declaration for `_1` is

```
using std::placeholders::_1;
```

This declaration says we're using the name `_1`, which is defined in the namespace `placeholders`, which is itself defined in the namespace `std`.

We must provide a separate `using` declaration for each placeholder name that we use. Writing such declarations can be tedious and error-prone. Rather than separately declaring each placeholder, we can use a different form of `using` that we will cover in more detail in § 18.2.2 (p. 793). This form:

```
using namespace namespace_name;
```

says that we want to make all the names from `namespace_name` accessible to our program. For example:

```
using namespace std::placeholders;
```

makes all the names defined by `placeholders` usable. Like the `bind` function, the `placeholders` namespace is defined in the functional header.

## Arguments to `bind`

As we've seen, we can use `bind` to fix the value of a parameter. More generally, we can use `bind` to bind or rearrange the parameters in the given callable. For example, assuming `f` is a callable object that has five parameters, the following call to `bind`:

```
// g is a callable object that takes two arguments
auto g = bind(f, a, b, _2, c, _1);
```

generates a new callable that takes two arguments, represented by the placeholders `_2` and `_1`. The new callable will pass its own arguments as the third and fifth arguments to `f`. The first, second, and fourth arguments to `f` are bound to the given values, `a`, `b`, and `c`, respectively.

The arguments to `g` are bound positionally to the placeholders. That is, the first argument to `g` is bound to `_1`, and the second argument is bound to `_2`. Thus, when we call `g`, the first argument to `g` will be passed as the last argument to `f`; the second argument to `g` will be passed as `f`'s third argument. In effect, this call to `bind` maps

```
g(_1, _2)
```

to

```
f(a, b, _2, c, _1)
```

That is, calling `g` calls `f` using `g`'s arguments for the placeholders along with the bound arguments, `a`, `b`, and `c`. For example, calling `g(x, y)` calls

```
f(a, b, y, c, x)
```

## Using to bind to Reorder Parameters

As a more concrete example of using bind to reorder arguments, we can use bind to invert the meaning of `isShorter` by writing

```
// sort on word length, shortest to longest
sort(words.begin(), words.end(), isShorter);
// sort on word length, longest to shortest
sort(words.begin(), words.end(), bind(isShorter, _2, _1));
```

In the first call, when `sort` needs to compare two elements, A and B, it will call `isShorter(A, B)`. In the second call to `sort`, the arguments to `isShorter` are swapped. In this case, when `sort` compares elements, it will be as if `sort` called `isShorter(B, A)`.

## Binding Reference Parameters

By default, the arguments to bind that are not placeholders are copied into the callable object that `bind` returns. However, as with lambdas, sometimes we have arguments that we want to bind but that we want to pass by reference or we might want to bind an argument that has a type that we cannot copy.

For example, to replace the lambda that captured an `ostream` by reference:

```
// os is a local variable referring to an output stream
// c is a local variable of type char
for_each(words.begin(), words.end(),
    [&os, c](const string &s) { os << s << c; });
```

We can easily write a function to do the same job:

```
ostream &print(ostream &os, const string &s, char c)
{
    return os << s << c;
}
```

However, we can't use `bind` directly to replace the capture of `os`:

```
// error: cannot copy os
for_each(words.begin(), words.end(), bind(print, os, _1, ' '));
```

because `bind` copies its arguments and we cannot copy an `ostream`. If we want to pass an object to `bind` without copying it, we must use the library `ref` function:

```
for_each(words.begin(), words.end(),
    bind(print, ref(os), _1, ' '));
```

The `ref` function returns an object that contains the given reference and that is itself copyable. There is also a  `cref` function that generates a class that holds a reference to `const`. Like `bind`, the `ref` and  `cref` functions are defined in the functional header.

**BACKWARD COMPATIBILITY: BINDING ARGUMENTS**

Older versions of C++ provided a much more limited, yet more complicated, set of facilities to bind arguments to functions. The library defined two functions named `bind1st` and `bind2nd`. Like `bind`, these functions take a function and generate a new callable object that calls the given function with one of its parameters bound to a given value. However, these functions can bind only the first or second parameter, respectively. Because they are of much more limited utility, they have been *deprecated* in the new standard. A deprecated feature is one that may not be supported in future releases. Modern C++ programs should use `bind`.

**EXERCISES SECTION 10.3.4**

**Exercise 10.22:** Rewrite the program to count words of size 6 or less using functions in place of the lambdas.

**Exercise 10.23:** How many arguments does `bind` take?

**Exercise 10.24:** Use `bind` and `check_size` to find the first element in a vector of `ints` that has a value greater than the length of a specified `string` value.

**Exercise 10.25:** In the exercises for § 10.3.2 (p. 392) you wrote a version of `biggies` that uses `partition`. Rewrite that function to use `check_size` and `bind`.

## 10.4 Revisiting Iterators

In addition to the iterators that are defined for each of the containers, the library defines several additional kinds of iterators in the `iterator` header. These iterators include

- **Insert iterators:** These iterators are bound to a container and can be used to insert elements into the container.
- **Stream iterators:** These iterators are bound to input or output streams and can be used to iterate through the associated IO stream.
- **Reverse iterators:** These iterators move backward, rather than forward. The library containers, other than `forward_list`, have reverse iterators.
- **Move iterators:** These special-purpose iterators move rather than copy their elements. We'll cover move iterators in § 13.6.2 (p. 543).

### 10.4.1 Insert Iterators



An inserter is an iterator adaptor (§ 9.6, p. 368) that takes a container and yields an iterator that adds elements to the specified container. When we assign a value through an inserter, the iterator calls a container operation to add an element at a specified position in the given container. The operations these iterators support are listed in Table 10.2 (overleaf).

There are three kinds of inserters. Each differs from the others as to where elements are inserted:

- `back_inserter` (§ 10.2.2, p. 382) creates an iterator that uses `push_back`.
- **`front_inserter`** creates an iterator that uses `push_front`.
- **`inserter`** creates an iterator that uses `insert`. This function takes a second argument, which must be an iterator into the given container. Elements are inserted ahead of the element denoted by the given iterator.



We can use `front_inserter` *only* if the container has `push_front`. Similarly, we can use `back_inserter` *only* if it has `push_back`.

**Table 10.2: Insert Iterator Operations**

<code>it = t</code>	Inserts the value <code>t</code> at the current position denoted by <code>it</code> . Depending on the kind of insert iterator, and assuming <code>c</code> is the container to which <code>it</code> is bound, calls <code>c.push_back(t)</code> , <code>c.push_front(t)</code> , or <code>c.insert(t, p)</code> , where <code>p</code> is the iterator position given to <code>inserter</code> .
<code>*it, ++it, it++</code>	These operations exist but do nothing to <code>it</code> . Each operator returns <code>it</code> .

It is important to understand that when we call `inserter(c, iter)`, we get an iterator that, when used successively, inserts elements ahead of the element originally denoted by `iter`. That is, if `it` is an iterator generated by `inserter`, then an assignment such as

`*it = val;`

behaves as

```
it = c.insert(it, val); // it points to the newly added element
++it; // increment it so that it denotes the same element as before
```

The iterator generated by `front_inserter` behaves quite differently from the one created by `inserter`. When we use `front_inserter`, elements are always inserted ahead of the then first element in the container. Even if the position we pass to `inserter` initially denotes the first element, as soon as we insert an element in front of that element, that element is no longer the one at the beginning of the container:

```
list<int> lst = {1,2,3,4};
list<int> lst2, lst3;           // empty lists
// after copy completes, lst2 contains 4 3 2 1
copy(lst.cbegin(), lst.cend(), front_inserter(lst2));
// after copy completes, lst3 contains 1 2 3 4
copy(lst.cbegin(), lst.cend(), inserter(lst3, lst3.begin()));
```

When we call `front_inserter(c)`, we get an insert iterator that successively calls `push_front`. As each element is inserted, it becomes the new first element in `c`. Therefore, `front_inserter` yields an iterator that reverses the order of the sequence that it inserts; `inserter` and `back_inserter` don't.

## EXERCISES SECTION 10.4.1

**Exercise 10.26:** Explain the differences among the three kinds of insert iterators.

**Exercise 10.27:** In addition to `unique` (§ 10.2.3, p. 384), the library defines function named `unique_copy` that takes a third iterator denoting a destination into which to copy the unique elements. Write a program that uses `unique_copy` to copy the unique elements from a `vector` into an initially empty list.

**Exercise 10.28:** Copy a `vector` that holds the values from 1 to 9 inclusive, into three other containers. Use an `inserter`, a `back_inserter`, and a `front_inserter`, respectively to add elements to these containers. Predict how the output sequence varies by the kind of inserter and verify your predictions by running your programs.

## 10.4.2 `iostream` Iterators



Even though the `iostream` types are not containers, there are iterators that can be used with objects of the IO types (§ 8.1, p. 310). An `istream_iterator` (Table 10.3 (overleaf)) reads an input stream, and an `ostream_iterator` (Table 10.4 (p. 405)) writes an output stream. These iterators treat their corresponding stream as a sequence of elements of a specified type. Using a stream iterator, we can use the generic algorithms to read data from or write data to stream objects.

### Operations on `istream_iterator`s

When we create a stream iterator, we must specify the type of objects that the iterator will read or write. An `istream_iterator` uses `>>` to read a stream. Therefore, the type that an `istream_iterator` reads must have an input operator defined. When we create an `istream_iterator`, we can bind it to a stream. Alternatively, we can default initialize the iterator, which creates an iterator that we can use as the off-the-end value.

```
istream_iterator<int> int_it(cin);    // reads ints from cin
istream_iterator<int> int_eof;        // end iterator value
ifstream in("afile");
istream_iterator<string> str_it(in); // reads strings from "afile"
```

As an example, we can use an `istream_iterator` to read the standard input into a `vector`:

```
istream_iterator<int> in_iter(cin); // read ints from cin
istream_iterator<int> eof;          // istream "end" iterator
while (in_iter != eof) // while there's valid input to read
    // postfix increment reads the stream and returns the old value of the iterator
    // we dereference that iterator to get the previous value read from the stream
    vec.push_back(*in_iter++);
```

This loop reads ints from `cin`, storing what was read in `vec`. On each iteration, the loop checks whether `in_iter` is the same as `eof`. That iterator was defined as the empty `istream_iterator`, which is used as the end iterator. An iterator

bound to a stream is equal to the end iterator once its associated stream hits end-of-file or encounters an IO error.

The hardest part of this program is the argument to `push_back`, which uses the dereference and postfix increment operators. This expression works just like others we've written that combined dereference with postfix increment (§ 4.5, p. 148). The postfix increment advances the stream by reading the next value but returns the *old* value of the iterator. That old value contains the previous value read from the stream. We dereference that iterator to obtain that value.

What is more useful is that we can rewrite this program as

```
istream_iterator<int> in_iter(cin), eof; // read ints from cin
vector<int> vec(in_iter, eof); // construct vec from an iterator range
```

Here we construct `vec` from a pair of iterators that denote a range of elements. Those iterators are `istream_iterators`, which means that the range is obtained by reading the associated stream. This constructor reads `cin` until it hits end-of-file or encounters an input that is not an `int`. The elements that are read are used to construct `vec`.

**Table 10.3: `istream_iterator` Operations**

<code>istream_iterator&lt;T&gt; in(is);</code>	<code>in</code> reads values of type <code>T</code> from input stream <code>is</code> .
<code>istream_iterator&lt;T&gt; end;</code>	Off-the-end iterator for an <code>istream_iterator</code> that reads values of type <code>T</code> .
<code>in == in2</code>	<code>in1</code> and <code>in2</code> must read the same type. They are equal if they are both the end value or are bound to the same input stream.
<code>in != in2</code>	
<code>*in</code>	Returns the value read from the stream.
<code>in-&gt;mem</code>	Synonym for <code>(*in).mem</code> .
<code>++in, in++</code>	Reads the next value from the input stream using the <code>&gt;&gt;</code> operator for the element type. As usual, the prefix version returns a reference to the incremented iterator. The postfix version returns the old value.

## Using Stream Iterators with the Algorithms

Because algorithms operate in terms of iterator operations, and the stream iterators support at least some iterator operations, we can use stream iterators with at least some of the algorithms. We'll see in § 10.5.1 (p. 410) how to tell which algorithms can be used with the stream iterators. As one example, we can call `accumulate` with a pair of `istream_iterators`:

```
istream_iterator<int> in(cin), eof;
cout << accumulate(in, eof, 0) << endl;
```

This call will generate the sum of values read from the standard input. If the input to this program is

```
23 109 45 89 6 34 12 90 34 23 56 23 8 89 23
```

then the output will be 664.

## **istream\_iterators Are Permitted to Use Lazy Evaluation**

When we bind an `istream_iterator` to a stream, we are not guaranteed that it will read the stream immediately. The implementation is permitted to delay reading the stream until we use the iterator. We are guaranteed that before we dereference the iterator for the first time, the stream will have been read. For most programs, whether the read is immediate or delayed makes no difference. However, if we create an `istream_iterator` that we destroy without using or if we are synchronizing reads to the same stream from two different objects, then we might care a great deal when the read happens.

## **Operations on ostream\_iterators**

An `ostream_iterator` can be defined for any type that has an output operator (the `<<` operator). When we create an `ostream_iterator`, we may (optionally) provide a second argument that specifies a character string to print following each element. That string must be a C-style character string (i.e., a string literal or a pointer to a null-terminated array). We must bind an `ostream_iterator` to a specific stream. There is no empty or off-the-end `ostream_iterator`.

**Table 10.4: ostream\_iterator Operations**

<code>ostream_iterator&lt;T&gt; out(os);</code>	out writes values of type T to output stream os.
<code>ostream_iterator&lt;T&gt; out(os, d);</code>	out writes values of type T followed by d to output stream os. d points to a null-terminated character array.
<code>out = val</code>	Writes val to the ostream to which out is bound using the <code>&lt;&lt;</code> operator. val must have a type that is compatible with the type that out can write.
<code>*out, ++out,</code> <code>out++</code>	These operations exist but do nothing to out. Each operator returns out.

We can use an `ostream_iterator` to write a sequence of values:

```
ostream_iterator<int> out_iter(cout, " ");
for (auto e : vec)
    *out_iter++ = e; // the assignment writes this element to cout
cout << endl;
```

This program writes each element from `vec` onto `cout` following each element with a space. Each time we assign a value to `out_iter`, the write is committed.

It is worth noting that we can omit the dereference and the increment when we assign to `out_iter`. That is, we can write this loop equivalently as

```
for (auto e : vec)
    out_iter = e; // the assignment writes this element to cout
cout << endl;
```

The `*` and `++` operators do nothing on an `ostream_iterator`, so omitting them has no effect on our program. However, we prefer to write the loop as first presented. That loop uses the iterator consistently with how we use other iterator

types. We can easily change this loop to execute on another iterator type. Moreover, the behavior of this loop will be clearer to readers of our code.

Rather than writing the loop ourselves, we can more easily print the elements in `vec` by calling `copy`:

```
copy(vec.begin(), vec.end(), out_iter);
cout << endl;
```

## Using Stream Iterators with Class Types

We can create an `istream_iterator` for any type that has an input operator (`>>`). Similarly, we can define an `ostream_iterator` so long as the type has an output operator (`<<`). Because `Sales_item` has both input and output operators, we can use IO iterators to rewrite the bookstore program from § 1.6 (p. 24):

```
istream_iterator<Sales_item> item_iter(cin), eof;
ostream_iterator<Sales_item> out_iter(cout, "\n");
// store the first transaction in sum and read the next record
Sales_item sum = *item_iter++;
while (item_iter != eof) {
    // if the current transaction (which is stored in item_iter) has the same ISBN
    if (item_iter->isbn() == sum.isbn())
        sum += *item_iter++; // add it to sum and read the next transaction
    else {
        out_iter = sum;      // write the current sum
        sum = *item_iter++; // read the next transaction
    }
}
out_iter = sum; // remember to print the last set of records
```

This program uses `item_iter` to read `Sales_item` transactions from `cin`. It uses `out_iter` to write the resulting sums to `cout`, following each output with a newline. Having defined our iterators, we use `item_iter` to initialize `sum` with the value of the first transaction:

```
// store the first transaction in sum and read the next record
Sales_item sum = *item_iter++;
```

Here, we dereference the result of the postfix increment on `item_iter`. This expression reads the next transaction, and initializes `sum` from the value previously stored in `item_iter`.

The `while` loop executes until we hit end-of-file on `cin`. Inside the `while`, we check whether `sum` and the record we just read refer to the same book. If so, we add the most recently read `Sales_item` into `sum`. If the ISBNs differ, we assign `sum` to `out_iter`, which prints the current value of `sum` followed by a newline. Having printed the sum for the previous book, we assign `sum` a copy of the most recently read transaction and increment the iterator, which reads the next transaction. The loop continues until an error or end-of-file is encountered. Before exiting, we remember to print the values associated with the last book in the input.

**EXERCISES SECTION 10.4.2**

**Exercise 10.29:** Write a program using stream iterators to read a text file into a vector of strings.

**Exercise 10.30:** Use stream iterators, `sort`, and `copy` to read a sequence of integers from the standard input, sort them, and then write them back to the standard output.

**Exercise 10.31:** Update the program from the previous exercise so that it prints only the unique elements. Your program should use `unique_copy` (§ 10.4.1, p. 403).

**Exercise 10.32:** Rewrite the bookstore problem from § 1.6 (p. 24) using a vector to hold the transactions and various algorithms to do the processing. Use `sort` with your `compareIsbn` function from § 10.3.1 (p. 387) to arrange the transactions in order, and then use `find` and `accumulate` to do the sum.

**Exercise 10.33:** Write a program that takes the names of an input file and two output files. The input file should hold integers. Using an `istream_iterator` read the input file. Using `ostream_iterators`, write the odd numbers into the first output file. Each value should be followed by a space. Write the even numbers into the second file. Each of these values should be placed on a separate line.

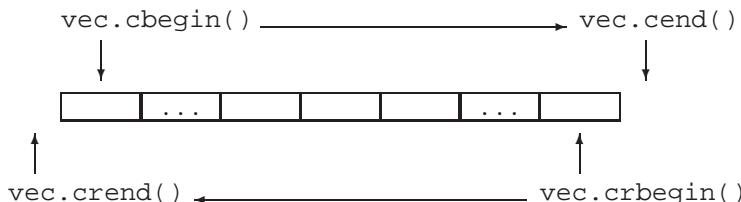
### 10.4.3 Reverse Iterators

A reverse iterator is an iterator that traverses a container backward, from the last element toward the first. A reverse iterator inverts the meaning of increment (and decrement). Incrementing (`++it`) a reverse iterator moves the iterator to the previous element; decrementing (`--it`) moves the iterator to the next element.

The containers, aside from `forward_list`, all have reverse iterators. We obtain a reverse iterator by calling the `rbegin`, `rend`, `crbegin`, and `crend` members. These members return reverse iterators to the last element in the container and one “past” (i.e., one before) the beginning of the container. As with ordinary iterators, there are both `const` and non`const` reverse iterators.

Figure 10.1 illustrates the relationship between these four iterators on a hypothetical vector named `vec`.

**Figure 10.1: Comparing begin/cend and rbegin/crend Iterators**



As an example, the following loop prints the elements of `vec` in reverse order:

```

vector<int> vec = {0,1,2,3,4,5,6,7,8,9};
// reverse iterator of vector from back to front
for (auto r_iter = vec.crbegin(); // binds r_iter to the last element
      r_iter != vec.crend(); // crend refers 1 before 1st element
      ++r_iter)             // decrements the iterator one element
    cout << *r_iter << endl; // prints 9, 8, 7, ... 0
  
```

Although it may seem confusing to have the meaning of the increment and decrement operators reversed, doing so lets us use the algorithms transparently to process a container forward or backward. For example, we can sort our vector in descending order by passing `sort` a pair of reverse iterators:

```

sort(vec.begin(), vec.end()); // sorts vec in "normal" order
// sorts in reverse: puts the smallest element at the end of vec
sort(vec.rbegin(), vec.rend());
  
```

## Reverse Iterators Require Decrement Operators

Not surprisingly, we can define a reverse iterator only from an iterator that supports `--` as well as `++`. After all, the purpose of a reverse iterator is to move the iterator backward through the sequence. Aside from `forward_list`, the iterators on the standard containers all support decrement as well as increment. However, the stream iterators do not, because it is not possible to move backward through a stream. Therefore, it is not possible to create a reverse iterator from a `forward_list` or a stream iterator.



## Relationship between Reverse Iterators and Other Iterators

Suppose we have a `string` named `line` that contains a comma-separated list of words, and we want to print the first word in `line`. Using `find`, this task is easy:

```

// find the first element in a comma-separated list
auto comma = find(line.cbegin(), line.cend(), ',');
cout << string(line.cbegin(), comma) << endl;
  
```

If there is a comma in `line`, then `comma` refers to that comma; otherwise it is `line.cend()`. When we print the `string` from `line.cbegin()` to `comma`, we print characters up to the comma, or the entire `string` if there is no comma.

If we wanted the last word, we can use reverse iterators instead:

```

// find the last element in a comma-separated list
auto rcomma = find(line.crbegin(), line.crend(), ',');
  
```

Because we pass `crbegin()` and `crend()`, this call starts with the last character in `line` and searches backward. When `find` completes, if there is a comma, then `rcomma` refers to the last comma in `line`—that is, it refers to the first comma found in the backward search. If there is no comma, then `rcomma` is `line.crend()`.

The interesting part comes when we try to print the word we found. The seemingly obvious way

```

// WRONG: will generate the word in reverse order
cout << string(line.crbegin(), rcomma) << endl;
  
```

generates bogus output. For example, had our input been

```
FIRST,MIDDLE,LAST
```

then this statement would print TSAL!

Figure 10.2 illustrates the problem: We are using reverse iterators, which process the string backward. Therefore, our output statement prints from `crbegin` backward through `line`. Instead, we want to print from `rcomma` forward to the end of `line`. However, we can't use `rcomma` directly. That iterator is a reverse iterator, which means that it goes backward toward the beginning of the string. What we need to do is transform `rcomma` back into an ordinary iterator that will go forward through `line`. We can do so by calling the `reverse_iterator`'s `base` member, which gives us its corresponding ordinary iterator:

```
// ok: get a forward iterator and read to the end of line
cout << string(rcomma.base(), line.cend()) << endl;
```

Given the same preceding input, this statement prints LAST as expected.

**Figure 10.2: Relationship between Reverse and Ordinary Iterators**



The objects shown in Figure 10.2 illustrate the relationship between ordinary and reverse iterators. For example, `rcomma` and `rcomma.base()` refer to different elements, as do `line.crbegin()` and `line.cend()`. These differences are needed to ensure that the *range* of elements, whether processed forward or backward, is the same.

Technically speaking, the relationship between normal and reverse iterators accommodates the properties of a left-inclusive range (§ 9.2.1, p. 331). The point is that `[line.crbegin(), rcomma]` and `[rcomma.base(), line.cend()]` refer to the same elements in `line`. In order for that to happen, `rcomma` and `rcomma.base()` must yield adjacent positions, rather than the same position, as must `crbegin()` and `cend()`.



The fact that reverse iterators are intended to represent ranges and that these ranges are asymmetric has an important consequence: When we initialize or assign a reverse iterator from a plain iterator, the resulting iterator does not refer to the same element as the original.

### EXERCISES SECTION 10.4.3

**Exercise 10.34:** Use `reverse_iterators` to print a vector in reverse order.

**Exercise 10.35:** Now print the elements in reverse order using ordinary iterators.

**Exercise 10.36:** Use `find` to find the last element in a list of ints with value 0.

**Exercise 10.37:** Given a vector that has ten elements, copy the elements from positions 3 through 7 in reverse order to a list.



## 10.5 Structure of Generic Algorithms

The most fundamental property of any algorithm is the list of operations it requires from its iterator(s). Some algorithms, such as `find`, require only the ability to access an element through the iterator, to increment the iterator, and to compare two iterators for equality. Others, such as `sort`, require the ability to read, write, and randomly access elements. The iterator operations required by the algorithms are grouped into five **iterator categories** listed in Table 10.5. Each algorithm specifies what kind of iterator must be supplied for each of its iterator parameters.

A second way is to classify the algorithms (as we did in the beginning of this chapter) is by whether they read, write, or reorder the elements in the sequence. Appendix A covers all the algorithms according to this classification.

The algorithms also share a set of parameter-passing conventions and a set of naming conventions, which we shall cover after looking at iterator categories.

**Table 10.5: Iterator Categories**

Input iterator	Read, but not write; single-pass, increment only
Output iterator	Write, but not read; single-pass, increment only
Forward iterator	Read and write; multi-pass, increment only
Bidirectional iterator	Read and write; multi-pass, increment and decrement
Random-access iterator	Read and write; multi-pass, full iterator arithmetic



### 10.5.1 The Five Iterator Categories

Like the containers, iterators define a common set of operations. Some operations are provided by all iterators; other operations are supported by only specific kinds of iterators. For example, `ostream_iterators` have only increment, dereference, and assignment. Iterators on `vector`, `strings`, and `deques` support these operations and the decrement, relational, and arithmetic operators.

Iterators are categorized by the operations they provide and the categories form a sort of hierarchy. With the exception of output iterators, an iterator of a higher category provides all the operations of the iterators of a lower categories.

The standard specifies the minimum category for each iterator parameter of the

generic and numeric algorithms. For example, `find`—which implements a one-pass, read-only traversal over a sequence—minimally requires an input iterator. The `replace` function requires a pair of iterators that are at least forward iterators. Similarly, `replace_copy` requires forward iterators for its first two iterators. Its third iterator, which represents a destination, must be at least an output iterator, and so on. For each parameter, the iterator must be at least as powerful as the stipulated minimum. Passing an iterator of a lesser power is an error.



Many compilers will not complain when we pass the wrong category of iterator to an algorithm.

## The Iterator Categories

**Input iterators:** can read elements in a sequence. An input iterator must provide

- Equality and inequality operators (`==`, `!=`) to compare two iterators
- Prefix and postfix increment (`++`) to advance the iterator
- Dereference operator (`*`) to read an element; dereference may appear only on the right-hand side of an assignment
- The arrow operator (`->`) as a synonym for `(*it).member`—that is, dereference the iterator and fetch a member from the underlying object

Input iterators may be used only sequentially. We are guaranteed that `*it++` is valid, but incrementing an input iterator may invalidate all other iterators into the stream. As a result, there is no guarantee that we can save the state of an input iterator and examine an element through that saved iterator. Input iterators, therefore, may be used only for single-pass algorithms. The `find` and `accumulate` algorithms require input iterators; `istream_iterator` are input iterators.

**Output iterators:** can be thought of as having complementary functionality to input iterators; they write rather than read elements. Output iterators must provide

- Prefix and postfix increment (`++`) to advance the iterator
- Dereference (`*`), which may appear only as the left-hand side of an assignment (Assigning to a dereferenced output iterator writes to the underlying element.)

We may assign to a given value of an output iterator only once. Like input iterators, output iterators may be used only for single-pass algorithms. Iterators used as a destination are typically output iterators. For example, the third parameter to `copy` is an output iterator. The `ostream_iterator` type is an output iterator.

**Forward iterators:** can read and write a given sequence. They move in only one direction through the sequence. Forward iterators support all the operations of both input iterators and output iterators. Moreover, they can read or write the same element multiple times. Therefore, we can use the saved state of a forward iterator. Hence, algorithms that use forward iterators may make multiple passes through the sequence. The `replace` algorithm requires a forward iterator; iterators on `forward_list` are forward iterators.

**Bidirectional iterators:** can read and write a sequence forward or backward. In addition to supporting all the operations of a forward iterator, a bidirectional iterator also supports the prefix and postfix decrement (--) operators. The `reverse` algorithm requires bidirectional iterators, and aside from `forward_list`, the library containers supply iterators that meet the requirements for a bidirectional iterator.

**Random-access iterators:** provide constant-time access to any position in the sequence. These iterators support all the functionality of bidirectional iterators. In addition, random-access iterators support the operations from Table 3.7 (p. 111):

- The relational operators (`<`, `<=`, `>`, and `>=`) to compare the relative positions of two iterators.
- Addition and subtraction operators (`+`, `+=`, `-`, and `-=`) on an iterator and an integral value. The result is the iterator advanced (or retreated) the integral number of elements within the sequence.
- The subtraction operator (`-`) when applied to two iterators, which yields the distance between two iterators.
- The subscript operator (`iter[n]`) as a synonym for `*(iter + n)`.

The `sort` algorithms require random-access iterators. Iterators for `array`, `deque`, `string`, and `vector` are random-access iterators, as are pointers when used to access elements of a built-in array.

### EXERCISES SECTION 10.5.1

**Exercise 10.38:** List the five iterator categories and the operations that each supports.

**Exercise 10.39:** What kind of iterator does a `list` have? What about a `vector`?

**Exercise 10.40:** What kinds of iterators do you think `copy` requires? What about `reverse` or `unique`?



## 10.5.2 Algorithm Parameter Patterns

Superimposed on any other classification of the algorithms is a set of parameter conventions. Understanding these parameter conventions can aid in learning new algorithms—by knowing what the parameters mean, you can concentrate on understanding the operation the algorithm performs. Most of the algorithms have one of the following four forms:

```
alg(beg, end, other args);
alg(beg, end, dest, other args);
alg(beg, end, beg2, other args);
alg(beg, end, beg2, end2, other args);
```

where `alg` is the name of the algorithm, and `beg` and `end` denote the input range on which the algorithm operates. Although nearly all algorithms take an input

range, the presence of the other parameters depends on the work being performed. The common ones listed here—`dest`, `beg2`, and `end2`—are all iterators. When used, these iterators fill similar roles. In addition to these iterator parameters, some algorithms take additional, noniterator parameters that are algorithm specific.

## Algorithms with a Single Destination Iterator

A `dest` parameter is an iterator that denotes a destination in which the algorithm can write its output. Algorithms *assume* that it is safe to write as many elements as needed.



Algorithms that write to an output iterator assume the destination is large enough to hold the output.

If `dest` is an iterator that refers directly to a container, then the algorithm writes its output to existing elements within the container. More commonly, `dest` is bound to an insert iterator (§ 10.4.1, p. 401) or an `ostream_iterator` (§ 10.4.2, p. 403). An insert iterator adds new elements to the container, thereby ensuring that there is enough space. An `ostream_iterator` writes to an output stream, again presenting no problem regardless of how many elements are written.

## Algorithms with a Second Input Sequence

Algorithms that take either `beg2` alone or `beg2` and `end2` use those iterators to denote a second input range. These algorithms typically use the elements from the second range in combination with the input range to perform a computation.

When an algorithm takes both `beg2` and `end2`, these iterators denote a second range. Such algorithms take two completely specified ranges: the input range denoted by `[beg, end]`, and a second input range denoted by `[beg2, end2]`.

Algorithms that take only `beg2` (and not `end2`) treat `beg2` as the first element in a second input range. The end of this range is not specified. Instead, these algorithms *assume* that the range starting at `beg2` is at least as large as the one denoted by `beg, end`.



Algorithms that take `beg2` alone *assume* that the sequence beginning at `beg2` is as large as the range denoted by `beg and end`.

### 10.5.3 Algorithm Naming Conventions

Separate from the parameter conventions, the algorithms also conform to a set of naming and overload conventions. These conventions deal with how we supply an operation to use in place of the default `<` or `==` operator and with whether the algorithm writes to its input sequence or to a separate destination.



#### Some Algorithms Use Overloading to Pass a Predicate

Algorithms that take a predicate to use in place of the `<` or `==` operator, and that do not take other arguments, typically are overloaded. One version of the function

uses the element type's operator to compare elements; the second takes an extra parameter that is a predicate to use in place of `<` or `==`:

```
unique(beg, end);           // uses the == operator to compare the elements
unique(beg, end, comp);    // uses comp to compare the elements
```

Both calls reorder the given sequence by removing adjacent duplicated elements. The first uses the element type's `==` operator to check for duplicates; the second calls `comp` to decide whether two elements are equal. Because the two versions of the function differ as to the number of arguments, there is no possible ambiguity (§ 6.4, p. 233) as to which function is being called.

## Algorithms with `_if` Versions

Algorithms that take an element value typically have a second named (not overloaded) version that takes a predicate (§ 10.3.1, p. 386) in place of the value. The algorithms that take a predicate have the suffix `_if` appended:

```
find(beg, end, val);      // find the first instance of val in the input range
find_if(beg, end, pred); // find the first instance for which pred is true
```

These algorithms both find the first instance of a specific element in the input range. The `find` algorithm looks for a specific value; the `find_if` algorithm looks for a value for which `pred` returns a nonzero value.

These algorithms provide a named version rather than an overloaded one because both versions of the algorithm take the same number of arguments. Overloading ambiguities would therefore be possible, albeit rare. To avoid any possible ambiguities, the library provides separate named versions for these algorithms.

## Distinguishing Versions That Copy from Those That Do Not

By default, algorithms that rearrange elements write the rearranged elements back into the given input range. These algorithms provide a second version that writes to a specified output destination. As we've seen, algorithms that write to a destination append `_copy` to their names (§ 10.2.2, p. 383):

```
reverse(beg, end);          // reverse the elements in the input range
reverse_copy(beg, end, dest); // copy elements in reverse order into dest
```

Some algorithms provide both `_copy` and `_if` versions. These versions take a destination iterator and a predicate:

```
// removes the odd elements from v1
remove_if(v1.begin(), v1.end(),
           [](int i) { return i % 2; });

// copies only the even elements from v1 into v2; v1 is unchanged
remove_copy_if(v1.begin(), v1.end(), back_inserter(v2),
               [](int i) { return i % 2; });
```

Both calls use a lambda (§ 10.3.2, p. 388) to determine whether an element is odd. In the first case, we remove the odd elements from the input sequence itself. In the second, we copy the non-odd (aka even) elements from the input range into `v2`.

### EXERCISES SECTION 10.5.3

**Exercise 10.41:** Based only on the algorithm and argument names, describe the operation that each of the following library algorithms performs:

```
replace(beg, end, old_val, new_val);
replace_if(beg, end, pred, new_val);
replace_copy(beg, end, dest, old_val, new_val);
replace_copy_if(beg, end, dest, pred, new_val);
```

## 10.6 Container-Specific Algorithms

Unlike the other containers, `list` and `forward_list` define several algorithms as members. In particular, the list types define their own versions of `sort`, `merge`, `remove`, `reverse`, and `unique`. The generic version of `sort` requires random-access iterators. As a result, `sort` cannot be used with `list` and `forward_list` because these types offer bidirectional and forward iterators, respectively.

The generic versions of the other algorithms that the list types define can be used with lists, but at a cost in performance. These algorithms swap elements in the input sequence. A list can “swap” its elements by changing the links among its elements rather than swapping the values of those elements. As a result, the list-specific versions of these algorithms can achieve much better performance than the corresponding generic versions.

These `list`-specific operations are described in Table 10.6. Generic algorithms not listed in the table that take appropriate iterators execute equally efficiently on `lists` and `forward_lists` as on other containers.



The list member versions should be used in preference to the generic algorithms for `lists` and `forward_lists`.

**Table 10.6: Algorithms That are Members of `list` and `forward_list`**

These operations return `void`.

<code>lst.merge(lst2)</code>	Merges elements from <code>lst2</code> onto <code>lst</code> . Both <code>lst</code> and <code>lst2</code> must be sorted. Elements are removed from <code>lst2</code> . After the <code>merge</code> , <code>lst2</code> is empty. The first version uses the <code>&lt;</code> operator; the second version uses the given comparison operation.
<code>lst.remove(val)</code>	Calls <code>erase</code> to remove each element that is <code>==</code> to the given value or for which the given unary predicate succeeds.
<code>lst.remove_if(pred)</code>	
<code>lst.reverse()</code>	Reverses the order of the elements in <code>lst</code> .
<code>lst.sort()</code>	Sorts the elements of <code>lst</code> using <code>&lt;</code> or the given comparison operation.
<code>lst.sort(comp)</code>	
<code>lst.unique()</code>	Calls <code>erase</code> to remove consecutive copies of the same value. The first version uses <code><code>=</code></code> ; the second uses the given binary predicate.
<code>lst.unique(pred)</code>	



## The `splice` Members

The list types also define a `splice` algorithm, which is described in Table 10.7. This algorithm is particular to list data structures. Hence a generic version of this algorithm is not needed.

**Table 10.7: Arguments to the `list` and `forward_list` `splice` Members**

<code>lst.splice(args)</code> or <code>flst.splice_after(args)</code>	
( <code>p</code> , <code>lst2</code> )	<code>p</code> is an iterator to an element in <code>lst</code> or an iterator just before an element in <code>flst</code> . Moves all the element(s) from <code>lst2</code> into <code>lst</code> just before <code>p</code> or into <code>flst</code> just after <code>p</code> . Removes the element(s) from <code>lst2</code> . <code>lst2</code> must have the same type as <code>lst</code> or <code>flst</code> and may not be the same list.
( <code>p</code> , <code>lst2</code> , <code>p2</code> )	<code>p2</code> is a valid iterator into <code>lst2</code> . Moves the element denoted by <code>p2</code> into <code>lst</code> or moves the element just after <code>p2</code> into <code>flst</code> . <code>lst2</code> can be the same list as <code>lst</code> or <code>flst</code> .
( <code>p</code> , <code>lst2</code> , <code>b</code> , <code>e</code> )	<code>b</code> and <code>e</code> must denote a valid range in <code>lst2</code> . Moves the elements in the given range from <code>lst2</code> . <code>lst2</code> and <code>lst</code> (or <code>flst</code> ) can be the same list but <code>p</code> must not denote an element in the given range.

## The List-Specific Operations Do Change the Containers

Most of the list-specific algorithms are similar—but not identical—to their generic counterparts. However, a crucially important difference between the list-specific and the generic versions is that the list versions change the underlying container. For example, the list version of `remove` removes the indicated elements. The list version of `unique` removes the second and subsequent duplicate elements.

Similarly, `merge` and `splice` are destructive on their arguments. For example, the generic version of `merge` writes the merged sequence to a given destination iterator; the two input sequences are unchanged. The list `merge` function destroys the given list—elements are removed from the argument list as they are merged into the object on which `merge` was called. After a `merge`, the elements from both lists continue to exist, but they are all elements of the same list.

### EXERCISES SECTION 10.6

**Exercise 10.42:** Reimplement the program that eliminated duplicate words that we wrote in § 10.2.3 (p. 383) to use a `list` instead of a `vector`.

*Associative containers* support efficient lookup and retrieval by a key. The two primary **associative-container** types are **map** and **set**. The elements in a map are key–value pairs: The key serves as an index into the map, and the value represents the data associated with that index. A set element contains only a key; a set supports efficient queries as to whether a given key is present. We might use a set to hold words that we want to ignore during some kind of text processing. A dictionary would be a good use for a map: The word would be the key, and its definition would be the value.

The library provides eight associative containers, listed in Table 11.1. These eight differ along three dimensions: Each container is (1) a set or a map, (2) requires unique keys or allows multiple keys, and (3) stores the elements in order or not. The containers that allow multiple keys include the word `multi`; those that do not keep their keys ordered start with the word `unordered`. Hence an `unordered_multiset` is a set that allows multiple keys whose elements are not stored in order, whereas a `set` has unique keys that are stored in order. The `unordered` containers use a hash function to organize their elements. We'll have more to say about the hash function in § 11.4 (p. 444).

The `map` and `multimap` types are defined in the `map` header; the `set` and `multiset` types are in the `set` header; and the `unordered` containers are in the `unordered_map` and `unordered_set` headers.

Table 11.1: Associative Container Types

Elements Ordered by Key	
<code>map</code>	Associative array; holds key–value pairs
<code>set</code>	Container in which the key is the value
<code>multimap</code>	map in which a key can appear multiple times
<code>multiset</code>	set in which a key can appear multiple times
Unordered Collections	
<code>unordered_map</code>	map organized by a hash function
<code>unordered_set</code>	set organized by a hash function
<code>unordered_multimap</code>	Hashed map; keys can appear multiple times
<code>unordered_multiset</code>	Hashed set; keys can appear multiple times



## 11.1 Using an Associative Container

Although most programmers are familiar with data structures such as `vectors` and `lists`, many have never used an associative data structure. Before we look at the details of how the library supports these types, it will be helpful to start with examples of how we can use these containers.

A map is a collection of key–value pairs. For example, each pair might contain a person's name as a key and a phone number as its value. We speak of such a data structure as "mapping names to phone numbers." The map type is often referred to as an **associative array**. An associative array is like a "normal" array except that its subscripts don't have to be integers. Values in a map are found by a key

rather than by their position. Given a `map` of names to phone numbers, we'd use a person's name as a subscript to fetch that person's phone number.

In contrast, a `set` is simply a collection of keys. A `set` is most useful when we simply want to know whether a value is present. For example, a business might define a `set` named `bad_checks` to hold the names of individuals who have written bad checks. Before accepting a check, that business would query `bad_checks` to see whether the customer's name was present.

## Using a `map`

A classic example that relies on associative arrays is a word-counting program:

```
// count the number of times each word occurs in the input
map<string, size_t> word_count; // empty map from string to size_t
string word;
while (cin >> word)
    ++word_count[word]; // fetch and increment the counter for word
for (const auto &w : word_count) // for each element in the map
    // print the results
    cout << w.first << " occurs " << w.second
    << ((w.second > 1) ? " times" : " time") << endl;
```

This program reads its input and reports how often each word appears.

Like the sequential containers, the associative containers are templates (§ 3.3, p. 96). To define a `map`, we must specify both the key and value types. In this program, the `map` stores elements in which the keys are `strings` and the values are `size_ts` (§ 3.5.2, p. 116). When we subscript `word_count`, we use a `string` as the subscript, and we get back the `size_t` counter associated with that `string`.

The `while` loop reads the standard input one word at a time. It uses each word to subscript `word_count`. If `word` is not already in the `map`, the subscript operator creates a new element whose key is `word` and whose value is 0. Regardless of whether the element had to be created, we increment the value.

Once we've read all the input, the range `for` (§ 3.2.3, p. 91) iterates through the `map`, printing each word and the corresponding counter. When we fetch an element from a `map`, we get an object of type `pair`, which we'll describe in § 11.2.3 (p. 426). Briefly, a `pair` is a template type that holds two (public) data elements named `first` and `second`. The pairs used by `map` have a `first` member that is the key and a `second` member that is the corresponding value. Thus, the effect of the output statement is to print each word and its associated counter.

If we ran this program on the text of the first paragraph in this section, our output would be

```
Although occurs 1 time
Before occurs 1 time
an occurs 1 time
and occurs 1 time
...
...
```

## Using a set

A logical extension to our program is to ignore common words like “the,” “and,” “or,” and so on. We’ll use a `set` to hold the words we want to ignore and count only those words that are not in this set:

```
// count the number of times each word occurs in the input
map<string, size_t> word_count; // empty map from string to size_t
set<string> exclude = {"The", "But", "And", "Or", "An", "A",
                       "the", "but", "and", "or", "an", "a"};
string word;
while (cin >> word)
    // count only words that are not in exclude
    if (exclude.find(word) == exclude.end())
        ++word_count[word]; // fetch and increment the counter for word
```

Like the other containers, `set` is a template. To define a `set`, we specify the type of its elements, which in this case are `strings`. As with the sequential containers, we can list initialize (§ 9.2.4, p. 336) the elements of an associative container. Our `exclude` set holds the 12 words we want to ignore.

The important difference between this program and the previous program is that before counting each word, we check whether the word is in the exclusion set. We do this check in the `if`:

```
// count only words that are not in exclude
if (exclude.find(word) == exclude.end())
```

The call to `find` returns an iterator. If the given key is in the `set`, the iterator refers to that key. If the element is not found, `find` returns the off-the-end iterator. In this version, we update the counter for `word` only if `word` is not in `exclude`.

If we run this version on the same input as before, our output would be

```
Although occurs 1 time
Before occurs 1 time
are occurs 1 time
as occurs 1 time
...
...
```

## EXERCISES SECTION 11.1

**Exercise 11.1:** Describe the differences between a `map` and a `vector`.

**Exercise 11.2:** Give an example of when each of `list`, `vector`, `deque`, `map`, and `set` might be most useful.

**Exercise 11.3:** Write your own version of the word-counting program.

**Exercise 11.4:** Extend your program to ignore case and punctuation. For example, “example.” “example,” and “Example” should all increment the same counter.

## 11.2 Overview of the Associative Containers

Associative containers (both ordered and unordered) support the general container operations covered in § 9.2 (p. 328) and listed in Table 9.2 (p. 330). The associative containers do *not* support the sequential-container position-specific operations, such as `push_front` or `back`. Because the elements are stored based on their keys, these operations would be meaningless for the associative containers. Moreover, the associative containers do not support the constructors or insert operations that take an element value and a count.

In addition to the operations they share with the sequential containers, the associative containers provide some operations (Table 11.7 (p. 438)) and type aliases (Table 11.3 (p. 429)) that the sequential containers do not. In addition, the unordered containers provide operations for tuning their hash performance, which we'll cover in § 11.4 (p. 444).

The associative container iterators are bidirectional (§ 10.5.1, p. 410).

### 11.2.1 Defining an Associative Container

As we've just seen, when we define a `map`, we must indicate both the key and value type; when we define a `set`, we specify only a key type, because there is no value type. Each of the associative containers defines a default constructor, which creates an empty container of the specified type. We can also initialize an associative container as a copy of another container of the same type or from a range of values, so long as those values can be converted to the type of the container. Under the new standard, we can also list initialize the elements:



C++  
11

```
map<string, size_t> word_count; // empty
// list initialization
set<string> exclude = { "the", "but", "and", "or", "an", "a",
                        "The", "But", "And", "Or", "An", "A" };
// three elements; authors maps last name to first
map<string, string> authors = { {"Joyce", "James"}, 
                                {"Austen", "Jane"}, 
                                {"Dickens", "Charles"} };
```

As usual, the initializers must be convertible to the type in the container. For `set`, the element type is the key type.

When we initialize a `map`, we have to supply both the key and the value. We wrap each key–value pair inside curly braces:

`{key, value}`

to indicate that the items together form one element in the `map`. The key is the first element in each pair, and the value is the second. Thus, `authors` maps last names to first names, and is initialized with three elements.

### Initializing a `multimap` or `multiset`

The keys in a `map` or a `set` must be unique; there can be only one element with a given key. The `multimap` and `multiset` containers have no such restriction;

there can be several elements with the same key. For example, the map we used to count words must have only one element per given word. On the other hand, a dictionary could have several definitions associated with a particular word.

The following example illustrates the differences between the containers with unique keys and those that have multiple keys. First, we'll create a vector of ints named `ivec` that has 20 elements: two copies of each of the integers from 0 through 9 inclusive. We'll use that vector to initialize a set and a multiset:

```
// define a vector with 20 elements, holding two copies of each number from 0 to 9
vector<int> ivec;
for (vector<int>::size_type i = 0; i != 10; ++i) {
    ivec.push_back(i);
    ivec.push_back(i); // duplicate copies of each number
}
// iset holds unique elements from ivec; miset holds all 20 elements
set<int> iset(ivec.cbegin(), ivec.cend());
multiset<int> miset(ivec.cbegin(), ivec.cend());
cout << ivec.size() << endl; // prints 20
cout << iset.size() << endl; // prints 10
cout << miset.size() << endl; // prints 20
```

Even though we initialized `iset` from the entire `ivec` container, `iset` has only ten elements: one for each distinct element in `ivec`. On the other hand, `miset` has 20 elements, the same as the number of elements in `ivec`.

## EXERCISES SECTION 11.2.1

**Exercise 11.5:** Explain the difference between a `map` and a `set`. When might you use one or the other?

**Exercise 11.6:** Explain the difference between a `set` and a `list`. When might you use one or the other?

**Exercise 11.7:** Define a `map` for which the key is the family's last name and the value is a vector of the children's names. Write code to add new families and to add new children to an existing family.

**Exercise 11.8:** Write a program that stores the excluded words in a `vector` instead of in a `set`. What are the advantages to using a `set`?



## 11.2.2 Requirements on Key Type

The associative containers place constraints on the type that is used as a key. We'll cover the requirements for keys in the unordered containers in § 11.4 (p. 445). For the ordered containers—`map`, `multimap`, `set`, and `multiset`—the key type must define a way to compare the elements. By default, the library uses the `<` operator for the key type to compare the keys. In the set types, the key is the element type;

in the map types, the key is the first type. Thus, the key type for `word_count` in § 11.1 (p. 421) is `string`. Similarly, the key type for `exclude` is `string`.



Callable objects passed to a sort algorithm (§ 10.3.1, p. 386) must meet the same requirements as do the keys in an associative container.

## Key Types for Ordered Containers

Just as we can provide our own comparison operation to an algorithm (§ 10.3, p. 385), we can also supply our own operation to use in place of the `<` operator on keys. The specified operation must define a **strict weak ordering** over the key type. We can think of a strict weak ordering as “less than,” although our function might use a more complicated procedure. However we define it, the comparison function must have the following properties:

- Two keys cannot both be “less than” each other; if `k1` is “less than” `k2`, then `k2` must never be “less than” `k1`.
- If `k1` is “less than” `k2` and `k2` is “less than” `k3`, then `k1` must be “less than” `k3`.
- If there are two keys, and neither key is “less than” the other, then we’ll say that those keys are “equivalent.” If `k1` is “equivalent” to `k2` and `k2` is “equivalent” to `k3`, then `k1` must be “equivalent” to `k3`.

If two keys are equivalent (i.e., if neither is “less than” the other), the container treats them as equal. When used as a key to a map, there will be only one element associated with those keys, and either key can be used to access the corresponding value.



In practice, what’s important is that a type that defines a `<` operator that “behaves normally” can be used as a key.

## Using a Comparison Function for the Key Type

The type of the operation that a container uses to organize its elements is part of the type of that container. To specify our own operation, we must supply the type of that operation when we define the type of an associative container. The operation type is specified following the element type inside the angle brackets that we use to say which type of container we are defining.

Each type inside the angle brackets is just that, a type. We supply a particular comparison operation (that must have the same type as we specified inside the angle brackets) as a constructor argument when we create a container.

For example, we can’t directly define a `multiset` of `Sales_data` because `Sales_data` doesn’t have a `<` operator. However, we can use the `compareIsbn` function from the exercises in § 10.3.1 (p. 387) to define a `multiset`. That function defines a strict weak ordering based on their ISBNs of two given `Sales_data` objects. The `compareIsbn` function should look something like

```
bool compareIsbn(const Sales_data &lhs, const Sales_data &rhs)
{
    return lhs.isbn() < rhs.isbn();
}
```

To use our own operation, we must define the `multiset` with two types: the key type, `Sales_data`, and the comparison type, which is a function pointer type (§ 6.7, p. 247) that can point to `compareIsbn`. When we define objects of this type, we supply a pointer to the operation we intend to use. In this case, we supply a pointer to `compareIsbn`:

```
// bookstore can have several transactions with the same ISBN
// elements in bookstore will be in ISBN order
multiset<Sales_data, decltype(compareIsbn)*>
bookstore(compareIsbn);
```

Here, we use `decltype` to specify the type of our operation, remembering that when we use `decltype` to form a function pointer, we must add a `*` to indicate that we're using a pointer to the given function type (§ 6.7, p. 250). We initialize `bookstore` from `compareIsbn`, which means that when we add elements to `bookstore`, those elements will be ordered by calling `compareIsbn`. That is, the elements in `bookstore` will be ordered by their ISBN members. We can write `compareIsbn` instead of `&compareIsbn` as the constructor argument because when we use the name of a function, it is automatically converted into a pointer if needed (§ 6.7, p. 248). We could have written `&compareIsbn` with the same effect.

### EXERCISES SECTION 11.2.2

**Exercise 11.9:** Define a `map` that associates words with a `list` of line numbers on which the word might occur.

**Exercise 11.10:** Could we define a `map` from `vector<int>::iterator` to `int`? What about from `list<int>::iterator` to `int`? In each case, if not, why not?

**Exercise 11.11:** Redefine `bookstore` without using `decltype`.

### 11.2.3 The `pair` Type

Before we look at the operations on associative containers, we need to know about the library type named `pair`, which is defined in the utility header.

A `pair` holds two data members. Like the containers, `pair` is a template from which we generate specific types. We must supply two type names when we create a `pair`. The data members of the `pair` have the corresponding types. There is no requirement that the two types be the same:

```
pair<string, string> anon;           // holds two strings
pair<string, size_t> word_count; // holds a string and an size_t
pair<string, vector<int>> line;   // holds string and vector<int>
```

The default pair constructor value initializes (§ 3.3.1, p. 98) the data members. Thus, `anon` is a pair of two empty strings, and `line` holds an empty string and an empty vector. The `size_t` value in `word_count` gets the value 0, and the `string` member is initialized to the empty string.

We can also provide initializers for each member:

```
pair<string, string> author{ "James", "Joyce" };
```

creates a pair named `author`, initialized with the values "James" and "Joyce".

**Table 11.2: Operations on pairs**

<code>pair&lt;T1, T2&gt; p;</code>	<code>p</code> is a pair with value initialized (§ 3.3.1, p. 98) members of types <code>T1</code> and <code>T2</code> , respectively.
<code>pair&lt;T1, T2&gt; p(v1, v2);</code>	<code>p</code> is a pair with types <code>T1</code> and <code>T2</code> ; the first and second members are initialized from <code>v1</code> and <code>v2</code> , respectively.
<code>pair&lt;T1, T2&gt; p = {v1, v2};</code>	Equivalent to <code>p(v1, v2)</code> .
<code>make_pair(v1, v2)</code>	Returns a pair initialized from <code>v1</code> and <code>v2</code> . The type of the pair is inferred from the types of <code>v1</code> and <code>v2</code> .
<code>p.first</code>	Returns the (public) data member of <code>p</code> named <code>first</code> .
<code>p.second</code>	Returns the (public) data member of <code>p</code> named <code>second</code> .
<code>p1 relop p2</code>	Relational operators ( <code>&lt;</code> , <code>&gt;</code> , <code>&lt;=</code> , <code>&gt;=</code> ). Relational operators are defined as dictionary ordering: For example, <code>p1 &lt; p2</code> is true if <code>p1.first &lt; p2.first</code> or if <code>!(p2.first &lt; p1.first) &amp;&amp; p1.second &lt; p2.second</code> . Uses the element's <code>&lt;</code> operator.
<code>p1 == p2</code>	Two pairs are equal if their <code>first</code> and <code>second</code> members are respectively equal. Uses the element's <code>==</code> operator.
<code>p1 != p2</code>	

Unlike other library types, the data members of `pair` are `public` (§ 7.2, p. 268). These members are named `first` and `second`, respectively. We access these members using the normal member access notation (§ 1.5.2, p. 23), as, for example, we did in the output statement of our word-counting program on page 421:

```
// print the results
cout << w.first << " occurs " << w.second
    << ((w.second > 1) ? " times" : " time") << endl;
```

Here, `w` is a reference to an element in a map. Elements in a map are pairs. In this statement we print the `first` member of the element, which is the key, followed by the `second` member, which is the counter. The library defines only a limited number of operations on pairs, which are listed in Table 11.2.

## A Function to Create pair Objects

Imagine we have a function that needs to return a `pair`. Under the new standard we can list initialize the return value (§ 6.3.2, p. 226):

```

pair<string, int>
process(vector<string> &v)
{
    // process v
    if (!v.empty())
        return {v.back(), v.back().size()}; // list initialize
    else
        return pair<string, int>(); // explicitly constructed return value
}

```

If `v` isn't empty, we return a pair composed of the last `string` in `v` and the size of that `string`. Otherwise, we explicitly construct and return an empty pair.

Under earlier versions of C++, we couldn't use braced initializers to return a type like `pair`. Instead, we might have written both returns to explicitly construct the return value:

```

if (!v.empty())
    return pair<string, int>(v.back(), v.back().size());

```

Alternatively, we could have used `make_pair` to generate a new `pair` of the appropriate type from its two arguments:

```

if (!v.empty())
    return make_pair(v.back(), v.back().size());

```

### EXERCISES SECTION 11.2.3

**Exercise 11.12:** Write a program to read a sequence of `strings` and `ints`, storing each into a pair. Store the pairs in a vector.

**Exercise 11.13:** There are at least three ways to create the pairs in the program for the previous exercise. Write three versions of that program, creating the pairs in each way. Explain which form you think is easiest to write and understand, and why.

**Exercise 11.14:** Extend the map of children to their family name that you wrote for the exercises in § 11.2.1 (p. 424) by having the vector store a pair that holds a child's name and birthday.

## 11.3 Operations on Associative Containers

In addition to the types listed in Table 9.2 (p. 330), the associative containers define the types listed in Table 11.3. These types represent the container's key and value types.

For the `set` types, the `key_type` and the `value_type` are the same; the values held in a `set` are the keys. In a `map`, the elements are key–value pairs. That is, each element is a `pair` object containing a key and a associated value. Because we cannot change an element's key, the key part of these pairs is `const`:

**Table 11.3: Associative Container Additional Type Aliases**

<code>key_type</code>	Type of the key for this container type
<code>mapped_type</code>	Type associated with each key; <b>map types only</b>
<code>value_type</code>	For sets, same as the <code>key_type</code> For maps, <code>pair&lt;const key_type, mapped_type&gt;</code>

```
set<string>::value_type v1;           // v1 is a string
set<string>::key_type v2;            // v2 is a string
map<string, int>::value_type v3;    // v3 is a pair<const string, int>
map<string, int>::key_type v4;      // v4 is a string
map<string, int>::mapped_type v5;   // v5 is an int
```

As with the sequential containers (§ 9.2.2, p. 332), we use the scope operator to fetch a type member—for example, `map<string, int>::key_type`.

Only the map types (`unordered_map`, `unordered_multimap`, `multimap`, and `map`) define `mapped_type`.

### 11.3.1 Associative Container Iterators

When we dereference an iterator, we get a reference to a value of the container's `value_type`. In the case of `map`, the `value_type` is a pair in which `first` holds the `const` key and `second` holds the value:

```
// get an iterator to an element in word_count
auto map_it = word_count.begin();
// *map_it is a reference to a pair<const string, size_t> object
cout << map_it->first;           // prints the key for this element
cout << " " << map_it->second; // prints the value of the element
map_it->first = "new key";       // error: key is const
++map_it->second;              // ok: we can change the value through an iterator
```



It is essential to remember that the `value_type` of a `map` is a pair and that we can change the value but not the key member of that pair.

### Iterators for sets Are `const`

Although the `set` types define both the iterator and `const_iterator` types, both types of iterators give us read-only access to the elements in the set. Just as we cannot change the key part of a `map` element, the keys in a `set` are also `const`. We can use a `set` iterator to read, but not write, an element's value:

```
set<int> iset = {0,1,2,3,4,5,6,7,8,9};
set<int>::iterator set_it = iset.begin();
if (set_it != iset.end()) {
    *set_it = 42;           // error: keys in a set are read-only
    cout << *set_it << endl; // ok: can read the key
}
```

## Iterating across an Associative Container

The map and set types provide all the begin and end operations from Table 9.2 (p. 330). As usual, we can use these functions to obtain iterators that we can use to traverse the container. For example, we can rewrite the loop that printed the results in our word-counting program on page 421 as follows:

```
// get an iterator positioned on the first element
auto map_it = word_count.cbegin();
// compare the current iterator to the off-the-end iterator
while (map_it != word_count.cend()) {
    // dereference the iterator to print the element key--value pairs
    cout << map_it->first << " occurs "
        << map_it->second << " times" << endl;
    ++map_it; // increment the iterator to denote the next element
}
```

The while condition and increment for the iterator in this loop look a lot like the programs we wrote that printed the contents of a vector or a string. We initialize an iterator, `map_it`, to refer to the first element in `word_count`. As long as the iterator is not equal to the end value, we print the current element and then increment the iterator. The output statement dereferences `map_it` to get the members of pair but is otherwise the same as the one in our original program.



The output of this program is in alphabetical order. When we use an iterator to traverse a map, multimap, set, or multiset, the iterators yield elements in ascending key order.

## Associative Containers and Algorithms

In general, we do not use the generic algorithms (Chapter 10) with the associative containers. The fact that the keys are `const` means that we cannot pass associative container iterators to algorithms that write to or reorder container elements. Such algorithms need to write to the elements. The elements in the `set` types are `const`, and those in maps are pairs whose first element is `const`.

Associative containers can be used with the algorithms that read elements. However, many of these algorithms search the sequence. Because elements in an associative container can be found (quickly) by their key, it is almost always a bad idea to use a generic search algorithm. For example, as we'll see in § 11.3.5 (p. 436), the associative containers define a member named `find`, which directly fetches the element with a given key. We could use the generic `find` algorithm to look for an element, but that algorithm does a sequential search. It is much faster to use the `find` member defined by the container than to call the generic version.

In practice, if we do so at all, we use an associative container with the algorithms either as the source sequence or as a destination. For example, we might use the generic `copy` algorithm to copy the elements from an associative container into another sequence. Similarly, we can call `inserter` to bind an insert iterator (§ 10.4.1, p. 401) to an associative container. Using `inserter`, we can use the associative container as a destination for another algorithm.

### EXERCISES SECTION 11.3.1

**Exercise 11.15:** What are the `mapped_type`, `key_type`, and `value_type` of a map from `int` to `vector<int>`?

**Exercise 11.16:** Using a map iterator write an expression that assigns a value to an element.

**Exercise 11.17:** Assuming `c` is a multiset of strings and `v` is a vector of strings, explain the following calls. Indicate whether each call is legal:

```
copy(v.begin(), v.end(), inserter(c, c.end()));
copy(v.begin(), v.end(), back_inserter(c));
copy(c.begin(), c.end(), inserter(v, v.end()));
copy(c.begin(), c.end(), back_inserter(v));
```

**Exercise 11.18:** Write the type of `map_it` from the loop on page 430 without using `auto` or `decltype`.

**Exercise 11.19:** Define a variable that you initialize by calling `begin()` on the multiset named `bookstore` from § 11.2.2 (p. 425). Write the variable's type without using `auto` or `decltype`.

## 11.3.2 Adding Elements

The `insert` members (Table 11.4 (overleaf)) add one element or a range of elements. Because `map` and `set` (and the corresponding unordered types) contain unique keys, inserting an element that is already present has no effect:

```
vector<int> ivec = {2,4,6,8,2,4,6,8};      // ivec has eight elements
set<int> set2;                            // empty set
set2.insert(ivec.cbegin(), ivec.cend());    // set2 has four elements
set2.insert({1,3,5,7,1,3,5,7});           // set2 now has eight elements
```

The versions of `insert` that take a pair of iterators or an initializer list work similarly to the corresponding constructors (§ 11.2.1, p. 423)—only the first element with a given key is inserted.

### Adding Elements to a map

When we insert into a `map`, we must remember that the element type is a pair. Often, we don't have a `pair` object that we want to insert. Instead, we create a `pair` in the argument list to `insert`:

```
// four ways to add word to word_count
word_count.insert({word, 1});
word_count.insert(make_pair(word, 1));
word_count.insert(pair<string, size_t>(word, 1));
word_count.insert(map<string, size_t>::value_type(word, 1));
```

As we've seen, under the new standard the easiest way to create a `pair` is to use brace initialization inside the argument list. Alternatively, we can call `make_pair`

or explicitly construct the pair. The argument in the last call to `insert`:

```
map<string, size_t>::value_type(s, 1)
```

constructs a new object of the appropriate pair type to insert into the map.

**Table 11.4: Associative Container `insert` Operations**

<code>c.insert(v)</code>	v <code>value_type</code> object; args are used to construct an element.
<code>c.emplace(args)</code>	For <code>map</code> and <code>set</code> , the element is inserted (or constructed) only if an element with the given key is not already in c. Returns a pair containing an iterator referring to the element with the given key and a bool indicating whether the element was inserted. For <code>multimap</code> and <code>multiset</code> , inserts (or constructs) the given element and returns an iterator to the new element.
<code>c.insert(b, e)</code>	b and e are iterators that denote a range of <code>c::value_type</code> values;
<code>c.insert(il)</code>	<code>il</code> is a braced list of such values. Returns void.
<code>c.insert(p, v)</code>	For <code>map</code> and <code>set</code> , inserts the elements with keys that are not already in c. For <code>multimap</code> and <code>multiset</code> inserts, each element in the range.
<code>c.emplace(p, args)</code>	Like <code>insert(v)</code> (or <code>emplace(args)</code> ), but uses iterator p as a hint for where to begin the search for where the new element should be stored. Returns an iterator to the element with the given key.

## Testing the Return from `insert`

The value returned by `insert` (or `emplace`) depends on the container type and the parameters. For the containers that have unique keys, the versions of `insert` and `emplace` that add a single element return a pair that lets us know whether the insertion happened. The first member of the pair is an iterator to the element with the given key; the second is a bool indicating whether that element was inserted, or was already there. If the key is already in the container, then `insert` does nothing, and the bool portion of the return value is false. If the key isn't present, then the element is inserted and the bool is true.

As an example, we'll rewrite our word-counting program to use `insert`:

```
// more verbose way to count number of times each word occurs in the input
map<string, size_t> word_count; // empty map from string to size_t
string word;
while (cin >> word) {
    // inserts an element with key equal to word and value 1;
    // if word is already in word_count, insert does nothing
    auto ret = word_count.insert({word, 1});
    if (!ret.second)           // word was already in word_count
        ++ret.first->second; // increment the counter
}
```

For each word, we attempt to `insert` it with a value 1. If word is already in the map, then nothing happens. In particular, the counter associated with word is

unchanged. If word is not already in the map, then that string is added to the map and its counter value is set to 1.

The if test examines the bool part of the return value. If that value is false, then the insertion didn't happen. In this case, word was already in word\_count, so we must increment the value associated with that element.

## Unwinding the Syntax

The statement that increments the counter in this version of the word-counting program can be hard to understand. It will be easier to understand that expression by first parenthesizing it to reflect the precedence (§ 4.1.2, p. 136) of the operators:

```
++( (ret.first)->second); // equivalent expression
```

Explaining this expression step by step:

**ret** holds the value returned by `insert`, which is a pair.

**ret.first** is the first member of that pair, which is a map iterator referring to the element with the given key.

**ret.first->** dereferences that iterator to fetch that element. Elements in the map are also pairs.

**ret.first->second** is the value part of the map element pair.

**++ret.first->second** increments that value.

Putting it back together, the increment statement fetches the iterator for the element with the key word and increments the counter associated with the key we tried to insert.

For readers using an older compiler or reading code that predates the new standard, declaring and initializing `ret` is also somewhat tricky:

```
pair<map<string, size_t>::iterator, bool> ret =  
    word_count.insert(make_pair(word, 1));
```

It should be easy to see that we're defining a pair and that the second type of the pair is `bool`. The first type of that pair is a bit harder to understand. It is the iterator type defined by the `map<string, size_t>` type.

## Adding Elements to `multiset` or `multimap`

Our word-counting program depends on the fact that a given key can occur only once. That way, there is only one counter associated with any given word. Sometimes, we want to be able to add additional elements with the same key. For example, we might want to map authors to titles of the books they have written. In this case, there might be multiple entries for each author, so we'd use a `multimap` rather than a `map`. Because keys in a multi container need not be unique, `insert` on these types always inserts an element:

```

multimap<string, string> authors;
// adds the first element with the key Barth, John
authors.insert({ "Barth, John", "Sot-Weed Factor" });
// ok: adds the second element with the key Barth, John
authors.insert({ "Barth, John", "Lost in the Funhouse" });

```

For the containers that allow multiple keys, the `insert` operation that takes a single element returns an iterator to the new element. There is no need to return a `bool`, because `insert` always adds a new element in these types.

### EXERCISES SECTION 11.3.2

**Exercise 11.20:** Rewrite the word-counting program from § 11.1 (p. 421) to use `insert` instead of subscripting. Which program do you think is easier to write and read? Explain your reasoning.

**Exercise 11.21:** Assuming `word_count` is a map from `string` to `size_t` and `word` is a `string`, explain the following loop:

```

while (cin >> word)
    ++word_count.insert({word, 0}).first->second;

```

**Exercise 11.22:** Given a `map<string, vector<int>>`, write the types used as an argument and as the return value for the version of `insert` that inserts one element.

**Exercise 11.23:** Rewrite the map that stored vectors of children's names with a key that is the family last name for the exercises in § 11.2.1 (p. 424) to use a `multimap`.

### 11.3.3 Erasing Elements

The associative containers define three versions of `erase`, which are described in Table 11.5. As with the sequential containers, we can `erase` one element or a range of elements by passing `erase` an iterator or an iterator pair. These versions of `erase` are similar to the corresponding operations on sequential containers: The indicated element(s) are removed and the function returns `void`.

The associative containers supply an additional `erase` operation that takes a `key_type` argument. This version removes all the elements, if any, with the given key and returns a count of how many elements were removed. We can use this version to remove a specific word from `word_count` before printing the results:

```

// erase on a key returns the number of elements removed
if (word_count.erase(removal_word))
    cout << "ok: " << removal_word << " removed\n";
else cout << "oops: " << removal_word << " not found!\n";

```

For the containers with unique keys, the return from `erase` is always either zero or one. If the return value is zero, then the element we wanted to `erase` was not in the container.

For types that allow multiple keys, the number of elements removed could be greater than one:

```
auto cnt = authors.erase("Barth, John");
```

If `authors` is the multimap we created in § 11.3.2 (p. 434), then `cnt` will be 2.

**Table 11.5: Removing Elements from an Associative Container**

<code>c.erase(k)</code>	Removes every element with key <code>k</code> from <code>c</code> . Returns <code>size_type</code> indicating the number of elements removed.
<code>c.erase(p)</code>	Removes the element denoted by the iterator <code>p</code> from <code>c</code> . <code>p</code> must refer to an actual element in <code>c</code> ; it must not be equal to <code>c.end()</code> . Returns an iterator to the element after <code>p</code> or <code>c.end()</code> if <code>p</code> denotes the last element in <code>c</code> .
<code>c.erase(b, e)</code>	Removes the elements in the range denoted by the iterator pair <code>b, e</code> . Returns <code>e</code> .

### 11.3.4 Subscripting a map



The `map` and `unordered_map` containers provide the subscript operator and a corresponding `at` function (§ 9.3.2, p. 348), which are described in Table 11.6 (overleaf). The `set` types do not support subscripting because there is no “value” associated with a key in a set. The elements are themselves keys, so the operation of “fetching the value associated with a key” is meaningless. We cannot subscript a `multimap` or an `unordered_multimap` because there may be more than one value associated with a given key.

Like the other subscript operators we’ve used, the `map` subscript takes an index (that is, a key) and fetches the value associated with that key. However, unlike other subscript operators, if the key is not already present, *a new element is created and inserted* into the `map` for that key. The associated value is value initialized (§ 3.3.1, p. 98).

For example, when we write

```
map <string, size_t> word_count; // empty map
// insert a value-initialized element with key Anna; then assign 1 to its value
word_count[ "Anna" ] = 1;
```

the following steps take place:

- `word_count` is searched for the element whose key is `Anna`. The element is not found.
- A new key-value pair is inserted into `word_count`. The key is a `const string` holding `Anna`. The value is value initialized, meaning in this case that the value is 0.
- The newly inserted element is fetched and is given the value 1.

Because the subscript operator might insert an element, we may use subscript only on a map that is not `const`.



Subscripting a map behaves quite differently from subscripting an array or vector: Using a key that is not already present *adds* an element with that key to the map.

**Table 11.6: Subscript Operation for `map` and `unordered_map`**

<code>c[k]</code>	Returns the element with key <code>k</code> ; if <code>k</code> is not in <code>c</code> , adds a new, value-initialized element with key <code>k</code> .
<code>c.at(k)</code>	Checked access to the element with key <code>k</code> ; throws an <code>out_of_range</code> exception (§ 5.6, p. 193) if <code>k</code> is not in <code>c</code> .

## Using the Value Returned from a Subscript Operation

Another way in which the `map` subscript differs from other subscript operators we've used is its return type. Ordinarily, the type returned by dereferencing an iterator and the type returned by the subscript operator are the same. Not so for maps: when we subscript a map, we get a `mapped_type` object; when we dereference a map iterator, we get a `value_type` object (§ 11.3, p. 428).

In common with other subscripts, the `map` subscript operator returns an lvalue (§ 4.1.1, p. 135). Because the return is an lvalue, we can read or write the element:

```
cout << word_count[ "Anna" ];    // fetch the element indexed by Anna; prints 1
++word_count[ "Anna" ];          // fetch the element and add 1 to it
cout << word_count[ "Anna" ];    // fetch the element and print it; prints 2
```



Unlike `vector` or `string`, the type returned by the `map` subscript operator differs from the type obtained by dereferencing a map iterator.

The fact that the subscript operator adds an element if it is not already in the map allows us to write surprisingly succinct programs such as the loop inside our word-counting program (§ 11.1, p. 421). On the other hand, sometimes we only want to know whether an element is present and *do not* want to add the element if it is not. In such cases, we must not use the subscript operator.

### 11.3.5 Accessing Elements

The associative containers provide various ways to find a given element, which are described in Table 11.7 (p. 438). Which operation to use depends on what problem we are trying to solve. If all we care about is whether a particular element is in the container, it is probably best to use `find`. For the containers that can hold only unique keys, it probably doesn't matter whether we use `find` or `count`. However, for the containers with multiple keys, `count` has to do more work: If the element

### EXERCISES SECTION 11.3.4

**Exercise 11.24:** What does the following program do?

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

**Exercise 11.25:** Contrast the following program with the one in the previous exercise

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

**Exercise 11.26:** What type can be used to subscript a map? What type does the subscript operator return? Give a concrete example—that is, define a map and then write the types that can be used to subscript the map and the type that would be returned from the subscript operator.

is present, it still has to count how many elements have the same key. If we don't need the count, it's best to use `find`:

```
set<int> iset = {0,1,2,3,4,5,6,7,8,9};  
iset.find(1); // returns an iterator that refers to the element with key == 1  
iset.find(11); // returns the iterator == iset.end()  
iset.count(1); // returns 1  
iset.count(11); // returns 0
```

### Using `find` Instead of Subscript for maps

For the `map` and `unordered_map` types, the subscript operator provides the simplest method of retrieving a value. However, as we've just seen, using a subscript has an important side effect: If that key is not already in the map, then subscript inserts an element with that key. Whether this behavior is correct depends on our expectations. Our word-counting programs relied on the fact that using a nonexistent key as a subscript inserts an element with that key and value 0.

Sometimes, we want to know if an element with a given key is present without changing the map. We cannot use the subscript operator to determine whether an element is present, because the subscript operator inserts a new element if the key is not already there. In such cases, we should use `find`:

```
if (word_count.find("foobar") == word_count.end())  
    cout << "foobar is not in the map" << endl;
```

### Finding Elements in a `multimap` or `multiset`

Finding an element in an associative container that requires unique keys is a simple matter—the element is or is not in the container. For the containers that allow multiple keys, the process is more complicated: There may be many elements with the given key. When a `multimap` or `multiset` has multiple elements of a given key, those elements will be adjacent within the container.

**Table 11.7: Operations to Find Elements in an Associative Container**

**lower\_bound** and **upper\_bound** not valid for the unordered containers.  
**Subscript** and **at** operations only for **map** and **unordered\_map** that are not **const**.

c.find(k)	Returns an iterator to the (first) element with key k, or the off-the-end iterator if k is not in the container.
c.count(k)	Returns the number of elements with key k. For the containers with unique keys, the result is always zero or one.
c.lower_bound(k)	Returns an iterator to the first element with key not less than k.
c.upper_bound(k)	Returns an iterator to the first element with key greater than k.
c.equal_range(k)	Returns a pair of iterators denoting the elements with key k. If k is not present, both members are c.end().

For example, given our map from author to titles, we might want to print all the books by a particular author. We can solve this problem in three different ways. The most obvious way uses `find` and `count`:

```
string search_item("Alain de Botton"); // author we'll look for
auto entries = authors.count(search_item); // number of elements
auto iter = authors.find(search_item); // first entry for this author
// loop through the number of entries there are for this author
while(entries) {
    cout << iter->second << endl; // print each title
    ++iter; // advance to the next title
    --entries; // keep track of how many we've printed
}
```

We start by determining how many entries there are for the author by calling `count` and getting an iterator to the first element with this key by calling `find`. The number of iterations of the `for` loop depends on the number returned from `count`. In particular, if the `count` was zero, then the loop is never executed.



We are guaranteed that iterating across a `multimap` or `multiset` returns all the elements with a given key in sequence.

## A Different, Iterator-Oriented Solution

Alternatively, we can solve our problem using `lower_bound` and `upper_bound`. Each of these operations take a key and returns an iterator. If the key is in the container, the iterator returned from `lower_bound` will refer to the first instance of that key and the iterator returned by `upper_bound` will refer just after the last instance of the key. If the element is not in the `multimap`, then `lower_bound` and `upper_bound` will return equal iterators; both will refer to the point at which the key can be inserted without disrupting the order. Thus, calling `lower_bound` and `upper_bound` on the same key yields an iterator range (§ 9.2.1, p. 331) that denotes all the elements with that key.

Of course, the iterator returned from these operations might be the off-the-end iterator for the container itself. If the element we're looking for has the largest key in the container, then `upper_bound` on that key returns the off-the-end iterator. If the key is not present and is larger than any key in the container, then the return from `lower_bound` will also be the off-the-end iterator.

 The iterator returned from `lower_bound` may or may not refer to an element with the given key. If the key is not in the container, then `lower_bound` refers to the first point at which this key can be inserted while preserving the element order within the container.

Using these operations, we can rewrite our program as follows:

```
// definitions of authors and search_item as above
// beg and end denote the range of elements for this author
for (auto beg = authors.lower_bound(search_item),
         end = authors.upper_bound(search_item);
     beg != end; ++beg)
    cout << beg->second << endl; // print each title
```

This program does the same work as the previous one that used `count` and `find` but accomplishes its task more directly. The call to `lower_bound` positions `beg` so that it refers to the first element matching `search_item` if there is one. If there is no such element, then `beg` refers to the first element with a key larger than `search_item`, which could be the off-the-end iterator. The call to `upper_bound` sets `end` to refer to the element just beyond the last element with the given key. These operations say nothing about whether the key is present. The important point is that the return values act like an iterator range (§ 9.2.1, p. 331).

If there is no element for this key, then `lower_bound` and `upper_bound` will be equal. Both will refer to the point at which this key can be inserted while maintaining the container order.

Assuming there are elements with this key, `beg` will refer to the first such element. We can increment `beg` to traverse the elements with this key. The iterator in `end` will signal when we've seen all the elements. When `beg` equals `end`, we have seen every element with this key.

Because these iterators form a range, we can use a `for` loop to traverse that range. The loop is executed zero or more times and prints the entries, if any, for the given author. If there are no elements, then `beg` and `end` are equal and the loop is never executed. Otherwise, we know that the increment to `beg` will eventually reach `end` and that in the process we will print each record associated with this author.

 If `lower_bound` and `upper_bound` return the same iterator, then the given key is not in the container.

## The `equal_range` Function

The remaining way to solve this problem is the most direct of the three approaches: Instead of calling `upper_bound` and `lower_bound`, we can call `equal_range`.

This function takes a key and returns a pair of iterators. If the key is present, then the first iterator refers to the first instance of the key and the second iterator refers one past the last instance of the key. If no matching element is found, then both the first and second iterators refer to the position where this key can be inserted.

We can use `equal_range` to modify our program once again:

```
// definitions of authors and search_item as above
// pos holds iterators that denote the range of elements for this key
for (auto pos = authors.equal_range(search_item);
     pos.first != pos.second; ++pos.first)
    cout << pos.first->second << endl; // print each title
```

This program is essentially identical to the previous one that used `upper_bound` and `lower_bound`. Instead of using local variables, `beg` and `end`, to hold the iterator range, we use the pair returned by `equal_range`. The first member of that pair holds the same iterator as `lower_bound` would have returned and `second` holds the iterator `upper_bound` would have returned. Thus, in this program `pos.first` is equivalent to `beg`, and `pos.second` is equivalent to `end`.

### EXERCISES SECTION 11.3.5

**Exercise 11.27:** What kinds of problems would you use `count` to solve? When might you use `find` instead?

**Exercise 11.28:** Define and initialize a variable to hold the result of calling `find` on a map from `string` to `vector<int>`.

**Exercise 11.29:** What do `upper_bound`, `lower_bound`, and `equal_range` return when you pass them a key that is not in the container?

**Exercise 11.30:** Explain the meaning of the operand `pos.first->second` used in the output expression of the final program in this section.

**Exercise 11.31:** Write a program that defines a multimap of authors and their works. Use `find` to find an element in the multimap and `erase` that element. Be sure your program works correctly if the element you look for is not in the map.

**Exercise 11.32:** Using the multimap from the previous exercise, write a program to print the list of authors and their works alphabetically.

### 11.3.6 A Word Transformation Map

We'll close this section with a program to illustrate creating, searching, and iterating across a map. We'll write a program that, given one `string`, transforms it into another. The input to our program is two files. The first file contains rules that we will use to transform the text in the second file. Each rule consists of a word that might be in the input file and a phrase to use in its place. The idea is that whenever the first word appears in the input, we will replace it with the corresponding phrase. The second file contains the text to transform.

If the contents of the word-transformation file are

```
brb be right back
k okay?
y why
r are
u you
pic picture
thk thanks!
18r later
```

and the text we are given to transform is

```
where r u
y dont u send me a pic
k thk 18r
```

then the program should generate the following output:

```
where are you
why dont you send me a picture
okay? thanks! later
```

## The Word Transformation Program

Our solution will use three functions. The `word_transform` function will manage the overall processing. It will take two `ifstream` arguments: The first will be bound to the word-transformation file and the second to the file of text we're to transform. The `buildMap` function will read the file of transformation rules and create a map from each word to its transformation. The `transform` function will take a `string` and return the transformation if there is one.

We'll start by defining the `word_transform` function. The important parts are the calls to `buildMap` and `transform`:

```
void word_transform(ifstream &map_file, ifstream &input)
{
    auto trans_map = buildMap(map_file); // store the transformations
    string text; // hold each line from the input
    while (getline(input, text)) { // read a line of input
        istringstream stream(text); // read each word
        string word;
        bool firstword = true; // controls whether a space is printed
        while (stream >> word) {
            if (firstword)
                firstword = false;
            else
                cout << " "; // print a space between words
            // transform returns its first argument or its transformation
            cout << transform(word, trans_map); // print the output
        }
        cout << endl; // done with this line of input
    }
}
```

The function starts by calling `buildMap` to generate the word-transformation map. We store the result in `trans_map`. The rest of the function processes the input file. The `while` loop uses `getline` to read the input file a line at a time. We read by line so that our output will have line breaks at the same position as in the input file. To get the words from each line, we use a nested `while` loop that uses an `istringstream` (§ 8.3, p. 321) to process each word in the current line.

The inner `while` prints the output using the `bool firstword` to determine whether to print a space. The call to `transform` obtains the word to print. The value returned from `transform` is either the original `string` in `word` or its corresponding transformation from `trans_map`.

## Building the Transformation Map

The `buildMap` function reads its given file and builds the transformation map.

```
map<string, string> buildMap(ifstream &map_file)
{
    map<string, string> trans_map; // holds the transformations
    string key; // a word to transform
    string value; // phrase to use instead
    // read the first word into key and the rest of the line into value
    while (map_file >> key && getline(map_file, value))
        if (value.size() > 1) // check that there is a transformation
            trans_map[key] = value.substr(1); // skip leading space
        else
            throw runtime_error("no rule for " + key);
    return trans_map;
}
```

Each line in `map_file` corresponds to a rule. Each rule is a word followed by a phrase, which might contain multiple words. We use `>>` to read the word that we will transform into `key` and call `getline` to read the rest of the line into `value`. Because `getline` does not skip leading spaces (§ 3.2.2, p. 87), we need to skip the space between the word and its corresponding rule. Before we store the transformation, we check that we got more than one character. If so, we call `substr` (§ 9.5.1, p. 361) to skip the space that separated the transformation phrase from its corresponding word and store that substring in `trans_map`,

Note that we use the subscript operator to add the key-value pairs. Implicitly, we are ignoring what should happen if a word appears more than once in our transformation file. If a word does appear multiple times, our loops will put the last corresponding phrase into `trans_map`. When the `while` concludes, `trans_map` contains the data that we need to transform the input.

## Generating a Transformation

The `transform` function does the actual transformation. Its parameters are references to the `string` to transform and to the transformation map. If the given `string` is in the map, `transform` returns the corresponding transformation. If the given `string` is not in the map, `transform` returns its argument:

```
const string &
transform(const string &s, const map<string, string> &m)
{
    // the actual map work; this part is the heart of the program
    auto map_it = m.find(s);
    // if this word is in the transformation map
    if (map_it != m.cend())
        return map_it->second; // use the replacement word
    else
        return s;                // otherwise return the original unchanged
}
```

We start by calling `find` to determine whether the given `string` is in the map. If it is, then `find` returns an iterator to the corresponding element. Otherwise, `find` returns the off-the-end iterator. If the element is found, we dereference the iterator, obtaining a pair that holds the key and value for that element (§ 11.3, p. 428). We return the second member, which is the transformation to use in place of `s`.

### EXERCISES SECTION 11.3.6

**Exercise 11.33:** Implement your own version of the word-transformation program.

**Exercise 11.34:** What would happen if we used the subscript operator instead of `find` in the `transform` function?

**Exercise 11.35:** In `buildMap`, what effect, if any, would there be from rewriting

```
trans_map[key] = value.substr(1);  
as trans_map.insert({key, value.substr(1)})?
```

**Exercise 11.36:** Our program does no checking on the validity of either input file. In particular, it assumes that the rules in the transformation file are all sensible. What would happen if a line in that file has a key, one space, and then the end of the line? Predict the behavior and then check it against your version of the program.

## 11.4 The Unordered Containers



The new standard defines four **unordered associative containers**. Rather than using a comparison operation to organize their elements, these containers use a **hash function** and the key type's `==` operator. An unordered container is most useful when we have a key type for which there is no obvious ordering relationship among the elements. These containers are also useful for applications in which the cost of maintaining the elements in order is prohibitive.



Although hashing gives better average case performance in principle, achieving good results in practice often requires a fair bit of performance testing and tweaking. As a result, it is usually easier (and often yields better performance) to use an ordered container.



Use an unordered container if the key type is inherently unordered or if performance testing reveals problems that hashing might solve.

## Using an Unordered Container

Aside from operations that manage the hashing, the unordered containers provide the same operations (`find`, `insert`, and so on) as the ordered containers. That means that the operations we've used on `map` and `set` apply to `unordered_map` and `unordered_set` as well. Similarly for the unordered versions of the containers that allow multiple keys.

As a result, we can usually use an unordered container in place of the corresponding ordered container, and vice versa. However, because the elements are not stored in order, the output of a program that uses an unordered container will (ordinarily) differ from the same program using an ordered container.

For example, we can rewrite our original word-counting program from § 11.1 (p. 421) to use an `unordered_map`:

```
// count occurrences, but the words won't be in alphabetical order
unordered_map<string, size_t> word_count;
string word;
while (cin >> word)
    ++word_count[word]; // fetch and increment the counter for word
for (const auto &w : word_count) // for each element in the map
    // print the results
    cout << w.first << " occurs " << w.second
    << ((w.second > 1) ? " times" : " time") << endl;
```

The type of `word_count` is the only difference between this program and our original. If we run this version on the same input as our original program,

```
containers. occurs 1 time
use occurs 1 time
can occurs 1 time
examples occurs 1 time
...
```

we'll obtain the same count for each word in the input. However, the output is unlikely to be in alphabetical order.

## Managing the Buckets

The unordered containers are organized as a collection of buckets, each of which holds zero or more elements. These containers use a hash function to map elements to buckets. To access an element, the container first computes the element's hash code, which tells which bucket to search. The container puts all of its elements with a given hash value into the same bucket. If the container allows multiple elements with a given key, all the elements with the same key will be in the same bucket. As a result, the performance of an unordered container depends on the quality of its hash function and on the number and size of its buckets.

The hash function must always yield the same result when called with the same argument. Ideally, the hash function also maps each particular value to a unique bucket. However, a hash function is allowed to map elements with differing keys to the same bucket. When a bucket holds several elements, those elements are searched sequentially to find the one we want. Typically, computing an element's hash code and finding its bucket is a fast operation. However, if the bucket has many elements, many comparisons may be needed to find a particular element.

The unordered containers provide a set of functions, listed in Table 11.8, that let us manage the buckets. These members let us inquire about the state of the container and force the container to reorganize itself as needed.

**Table 11.8: Unordered Container Management Operations**

<b>Bucket Interface</b>	
<code>c.bucket_count()</code>	Number of buckets in use.
<code>c.max_bucket_count()</code>	Largest number of buckets this container can hold.
<code>c.bucket_size(n)</code>	Number of elements in the nth bucket.
<code>c.bucket(k)</code>	Bucket in which elements with key k would be found.
<b>Bucket Iteration</b>	
<code>local_iterator</code>	Iterator type that can access elements in a bucket.
<code>const_local_iterator</code>	const version of the bucket iterator.
<code>c.begin(n), c.end(n)</code>	Iterator to the first, one past the last element in bucket n.
<code>c.cbegin(n), c.cend(n)</code>	Returns <code>const_local_iterator</code> .
<b>Hash Policy</b>	
<code>c.load_factor()</code>	Average number of elements per bucket. Returns float.
<code>c.max_load_factor()</code>	Average bucket size that c tries to maintain. c adds buckets to keep <code>load_factor &lt;= max_load_factor</code> . Returns float.
<code>c.rehash(n)</code>	Reorganize storage so that <code>bucket_count &gt;= n</code> and <code>bucket_count &gt; size/max_load_factor</code> .
<code>c.reserve(n)</code>	Reorganize so that c can hold n elements without a rehash.

## Requirements on Key Type for Unordered Containers

By default, the unordered containers use the `==` operator on the key type to compare elements. They also use an object of type `hash<key_type>` to generate the hash code for each element. The library supplies versions of the `hash` template for the built-in types, including pointers. It also defines `hash` for some of the library types, including `strings` and the smart pointer types that we will describe in Chapter 12. Thus, we can directly define unordered containers whose key is one of the built-in types (including pointer types), or a `string`, or a smart pointer.

However, we cannot directly define an unordered container that uses our own class types for its key type. Unlike the containers, we cannot use the `hash` template directly. Instead, we must supply our own version of the `hash` template. We'll see how to do so in § 16.5 (p. 709).

Instead of using the default `hash`, we can use a strategy similar to the one we used to override the default comparison operation on keys for the ordered

containers (§ 11.2.2, p. 425). To use `Sales_data` as the key, we'll need to supply functions to replace both the `==` operator and to calculate a hash code. We'll start by defining these functions:

```
size_t hasher(const Sales_data &sd)
{
    return hash<string>()(sd.isbn());
}
bool eqOp(const Sales_data &lhs, const Sales_data &rhs)
{
    return lhs.isbn() == rhs.isbn();
}
```

Our `hasher` function uses an object of the library `hash` of `string` type to generate a hash code from the `ISBN` member. Similarly, the `eqOp` function compares two `Sales_data` objects by comparing their ISBNs.

We can use these functions to define an `unordered_multiset` as follows

```
using SD_multiset = unordered_multiset<Sales_data,
                                         decltype(hasher)*, decltype(eqOp)*>;
// arguments are the bucket size and pointers to the hash function and equality operator
SD_multiset bookstore(42, hasher, eqOp);
```

To simplify the declaration of `bookstore` we first define a type alias (§ 2.5.1, p. 67) for an `unordered_multiset` whose hash and equality operations have the same types as our `hasher` and `eqOp` functions. Using that type, we define `bookstore` passing pointers to the functions we want `bookstore` to use.

If our class has its own `==` operator we can override just the hash function:

```
// use FooHash to generate the hash code; Foo must have an == operator
unordered_set<Foo, decltype(FooHash)*> fooSet(10, FooHash);
```

## EXERCISES SECTION 11.4

**Exercise 11.37:** What are the advantages of an unordered container as compared to the ordered version of that container? What are the advantages of the ordered version?

**Exercise 11.38:** Rewrite the word-counting (§ 11.1, p. 421) and word-transformation (§ 11.3.6, p. 440) programs to use an `unordered_map`.

### 12.3.1 Design of the Query Program



A good way to start the design of a program is to list the program's operations. Knowing what operations we need can help us see what data structures we'll need. Starting from requirements, the tasks our program must do include the following:

- When it reads the input, the program must remember the line(s) in which each word appears. Hence, the program will need to read the input a line at a time and break up the lines from the input file into its separate words
- When it generates output,
  - The program must be able to fetch the line numbers associated with a given word
  - The line numbers must appear in ascending order with no duplicates
  - The program must be able to print the text appearing in the input file at a given line number.

These requirements can be met quite neatly by using various library facilities:

- We'll use a `vector<string>` to store a copy of the entire input file. Each line in the input file will be an element in this `vector`. When we want to print a line, we can fetch the line using its line number as the index.
- We'll use an `istringstream` (§ 8.3, p. 321) to break each line into words.
- We'll use a `set` to hold the line numbers on which each word in the input appears. Using a `set` guarantees that each line will appear only once and that the line numbers will be stored in ascending order.
- We'll use a `map` to associate each word with the `set` of line numbers on which the word appears. Using a `map` will let us fetch the `set` for any given word.

For reasons we'll explain shortly, our solution will also use `shared_ptr`s.

## Data Structures

Although we could write our program using `vector`, `set`, and `map` directly, it will be more useful if we define a more abstract solution. We'll start by designing a class to hold the input file in a way that makes querying the file easy. This class, which we'll name `TextQuery`, will hold a `vector` and a `map`. The `vector` will hold the text of the input file; the `map` will associate each word in that file to the `set` of line numbers on which that word appears. This class will have a constructor that reads a given input file and an operation to perform the queries.

The work of the query operation is pretty simple: It will look inside its `map` to see whether the given word is present. The hard part in designing this function is deciding what the query function should return. Once we know that a word was found, we need to know how often it occurred, the line numbers on which it occurred, and the corresponding text for each of those line numbers.

The easiest way to return all those data is to define a second class, which we'll name `QueryResult`, to hold the results of a query. This class will have a `print` function to print the results in a `QueryResult`.

## Sharing Data between Classes

Our `QueryResult` class is intended to represent the results of a query. Those results include the set of line numbers associated with the given word and the corresponding lines of text from the input file. These data are stored in objects of type `TextQuery`.

Because the data that a `QueryResult` needs are stored in a `TextQuery` object, we have to decide how to access them. We could copy the set of line numbers, but that might be an expensive operation. Moreover, we certainly wouldn't want to copy the vector, because that would entail copying the entire file in order to print (what will usually be) a small subset of the file.

We could avoid making copies by returning iterators (or pointers) into the `TextQuery` object. However, this approach opens up a pitfall: What happens if the `TextQuery` object is destroyed before a corresponding `QueryResult`? In that case, the `QueryResult` would refer to data in an object that no longer exists.

This last observation about synchronizing the lifetime of a `QueryResult` with the `TextQuery` object whose results it represents suggests a solution to our design problem. Given that these two classes conceptually “share” data, we'll use `shared_ptr` (§ 12.1.1, p. 450) to reflect that sharing in our data structures.

## Using the `TextQuery` Class

When we design a class, it can be helpful to write programs using the class before actually implementing the members. That way, we can see whether the class has the operations we need. For example, the following program uses our proposed `TextQuery` and `QueryResult` classes. This function takes an `ifstream` that points to the file we want to process, and interacts with a user, printing the results for the given words:

```
void runQueries(ifstream &infile)
{
    // infile is an ifstream that is the file we want to query
    TextQuery tq(infile); // store the file and build the query map
    // iterate with the user: prompt for a word to find and print results
    while (true) {
        cout << "enter word to look for, or q to quit: ";
        string s;
        // stop if we hit end-of-file on the input or if a 'q' is entered
        if (!(cin >> s) || s == "q") break;
        // run the query and print the results
        print(cout, tq.query(s)) << endl;
    }
}
```

We start by initializing a `TextQuery` object named `tq` from a given `ifstream`. The `TextQuery` constructor reads that file into its `vector` and builds the `map` that associates the words in the input with the line numbers on which they appear.

The `while` loop iterates (indefinitely) with the user asking for a word to query and printing the related results. The loop condition tests the literal `true` (§ 2.1.3, p. 41), so it always succeeds. We exit the loop through the `break` (§ 5.5.1, p. 190)

after the first `if`. That `if` checks that the read succeeded. If so, it also checks whether the user entered a `q` to quit. Once we have a word to look for, we ask `tq` to find that word and then call `print` to print the results of the search.

### EXERCISES SECTION 12.3.1

**Exercise 12.27:** The `TextQuery` and `QueryResult` classes use only capabilities that we have already covered. Without looking ahead, write your own versions of these classes.

**Exercise 12.28:** Write a program to implement text queries without defining classes to manage the data. Your program should take a file and interact with a user to query for words in that file. Use `vector`, `map`, and `set` containers to hold the data for the file and to generate the results for the queries.

**Exercise 12.29:** We could have written the loop to manage the interaction with the user as a `do while` (§ 5.4.4, p. 189) loop. Rewrite the loop to use a `do while`. Explain which version you prefer and why.

### 12.3.2 Defining the Query Program Classes



We'll start by defining our `TextQuery` class. The user will create objects of this class by supplying an `istream` from which to read the input file. This class also provides the `query` operation that will take a `string` and return a `QueryResult` representing the lines on which that `string` appears.

The data members of the class have to take into account the intended sharing with `QueryResult` objects. The `QueryResult` class will share the `vector` representing the input file and the sets that hold the line numbers associated with each word in the input. Hence, our class has two data members: a `shared_ptr` to a dynamically allocated `vector` that holds the input file, and a `map` from `string` to `shared_ptr<set>`. The `map` associates each word in the file with a dynamically allocated `set` that holds the line numbers on which that word appears.

To make our code a bit easier to read, we'll also define a type member (§ 7.3.1, p. 271) to refer to line numbers, which are indices into a `vector` of `strings`:

```
class QueryResult; // declaration needed for return type in the query function
class TextQuery {
public:
    using line_no = std::vector<std::string>::size_type;
    TextQuery(std::ifstream&);
    QueryResult query(const std::string&) const;
private:
    std::shared_ptr<std::vector<std::string>> file; // input file
    // map of each word to the set of the lines in which that word appears
    std::map<std::string,
            std::shared_ptr<std::set<line_no>>> wm;
};
```

The hardest part about this class is untangling the class names. As usual, for code that will go in a header file, we use `std::` when we use a library name (§ 3.1, p. 83). In this case, the repeated use of `std::` makes the code a bit hard to read at first. For example,

```
std::map<std::string, std::shared_ptr<std::set<line_no>>> wm;
```

is easier to understand when rewritten as

```
map<string, shared_ptr<set<line_no>>> wm;
```

## The TextQuery Constructor

The `TextQuery` constructor takes an `ifstream`, which it reads a line at a time:

```
// read the input file and build the map of lines to line numbers
TextQuery::TextQuery(ifstream &is): file(new vector<string>)
{
    string text;
    while (getline(is, text)) {           // for each line in the file
        file->push_back(text);          // remember this line of text
        int n = file->size() - 1;        // the current line number
        istringstream line(text);         // separate the line into words
        string word;
        while (line >> word) {          // for each word in that line
            // if word isn't already in wm, subscripting adds a new entry
            auto &lines = wm[word]; // lines is a shared_ptr
            if (!lines) // that pointer is null the first time we see word
                lines.reset(new set<line_no>); // allocate a new set
            lines->insert(n);           // insert this line number
        }
    }
}
```

The constructor initializer allocates a new vector to hold the text from the input file. We use `getline` to read the file a line at a time and push each line onto the vector. Because `file` is a `shared_ptr`, we use the `->` operator to dereference `file` to fetch the `push_back` member of the vector to which `file` points.

Next we use an `istringstream` (§ 8.3, p. 321) to process each word in the line we just read. The inner `while` uses the `istringstream` input operator to read each word from the current line into `word`. Inside the `while`, we use the `map` subscript operator to fetch the `shared_ptr<set>` associated with `word` and bind `lines` to that pointer. Note that `lines` is a reference, so changes made to `lines` will be made to the element in `wm`.

If `word` wasn't in the `map`, the subscript operator adds `word` to `wm` (§ 11.3.4, p. 435). The element associated with `word` is value initialized, which means that `lines` will be a null pointer if the subscript operator added `word` to `wm`. If `lines` is null, we allocate a new `set` and call `reset` to update the `shared_ptr` to which `lines` refers to point to this newly allocated `set`.

Regardless of whether we created a new `set`, we call `insert` to add the current line number. Because `lines` is a reference, the call to `insert` adds an element

to the set in `wm`. If a given word occurs more than once in the same line, the call to `insert` does nothing.

## The `QueryResult` Class

The `QueryResult` class has three data members: a `string` that is the word whose results it represents; a `shared_ptr` to the vector containing the input file; and a `shared_ptr` to the set of line numbers on which this word appears. Its only member function is a constructor that initializes these three members:

```
class QueryResult {
    friend std::ostream& print(std::ostream&, const QueryResult&);
public:
    QueryResult(std::string s,
                std::shared_ptr<std::set<line_no>> p,
                std::shared_ptr<std::vector<std::string>> f):
        sought(s), lines(p), file(f) { }
private:
    std::string sought; // word this query represents
    std::shared_ptr<std::set<line_no>> lines; // lines it's on
    std::shared_ptr<std::vector<std::string>> file; // input file
};
```

The constructor's only job is to store its arguments in the corresponding data members, which it does in the constructor initializer list (§ 7.1.4, p. 265).

## The `query` Function

The `query` function takes a `string`, which it uses to locate the corresponding set of line numbers in the map. If the `string` is found, the `query` function constructs a `QueryResult` from the given `string`, the `TextQuery` `file` member, and the `set` that was fetched from `wm`.

The only question is: What should we return if the given `string` is not found? In this case, there is no `set` to return. We'll solve this problem by defining a local `static` object that is a `shared_ptr` to an empty set of line numbers. When the word is not found, we'll return a copy of this `shared_ptr`:

```
QueryResult
TextQuery::query(const string &sought) const
{
    // we'll return a pointer to this set if we don't find sought
    static shared_ptr<set<line_no>> nodata(new set<line_no>);
    // use find and not a subscript to avoid adding words to wm!
    auto loc = wm.find(sought);
    if (loc == wm.end())
        return QueryResult(sought, nodata, file); // not found
    else
        return QueryResult(sought, loc->second, file);
}
```

## Printing the Results

The print function prints its given `QueryResult` object on its given stream:

```
ostream &print(ostream & os, const QueryResult &qr)
{
    // if the word was found, print the count and all occurrences
    os << qr.sought << " occurs " << qr.lines->size() << " "
        << make_plural(qr.lines->size(), "time", "s") << endl;
    // print each line in which the word appeared
    for (auto num : *qr.lines) // for every element in the set
        // don't confound the user with text lines starting at 0
        os << "\t(line " << num + 1 << ") "
            << *(qr.file->begin() + num) << endl;
    return os;
}
```

We use the `size` of the set to which the `qr.lines` points to report how many matches were found. Because that set is in a `shared_ptr`, we have to remember to dereference `lines`. We call `make_plural` (§ 6.3.2, p. 224) to print `time` or `times`, depending on whether that size is equal to 1.

In the `for` we iterate through the set to which `lines` points. The body of the `for` prints the line number, adjusted to use human-friendly counting. The numbers in the set are indices of elements in the vector, which are numbered from zero. However, most users think of the first line as line number 1, so we systematically add 1 to the line numbers to convert to this more common notation.

We use the line number to fetch a line from the vector to which `file` points. Recall that when we add a number to an iterator, we get the element that many elements further into the vector (§ 3.4.2, p. 111). Thus, `file->begin() + num` is the `num`th element after the start of the vector to which `file` points.

Note that this function correctly handles the case that the word is not found. In this case, the set will be empty. The first output statement will note that the word occurred 0 times. Because `*res.lines` is empty, the `for` loop won't be executed.

### EXERCISES SECTION 12.3.2

**Exercise 12.30:** Define your own versions of the `TextQuery` and `QueryResult` classes and execute the `runQueries` function from § 12.3.1 (p. 486).

**Exercise 12.31:** What difference(s) would it make if we used a `vector` instead of a `set` to hold the line numbers? Which approach is better? Why?

**Exercise 12.32:** Rewrite the `TextQuery` and `QueryResult` classes to use a `StrBlob` instead of a `vector<string>` to hold the input file.

**Exercise 12.33:** In Chapter 15 we'll extend our query system and will need some additional members in the `QueryResult` class. Add members named `begin` and `end` that return iterators into the set of line numbers returned by a given query, and a member named `get_file` that returns a `shared_ptr` to the file in the `QueryResult` object.

*Many applications* include concepts that are related to but slightly different from one another. For example, our bookstore might offer different pricing strategies for different books. Some books might be sold only at a given price. Others might be sold subject to a discount. We might give a discount to purchasers who buy a specified number of copies of the book. Or we might give a discount for only the first few copies purchased but charge full price for any bought beyond a given limit, and so on. Object-oriented programming (OOP) is a good match to this kind of application.



## 15.1 OOP: An Overview

The key ideas in **object-oriented programming** are data abstraction, inheritance, and dynamic binding. Using data abstraction, we can define classes that separate interface from implementation (Chapter 7). Through inheritance, we can define classes that model the relationships among similar types. Through dynamic binding, we can use objects of these types while ignoring the details of how they differ.

### Inheritance

Classes related by **inheritance** form a hierarchy. Typically there is a **base class** at the root of the hierarchy, from which the other classes inherit, directly or indirectly. These inheriting classes are known as **derived classes**. The base class defines those members that are common to the types in the hierarchy. Each derived class defines those members that are specific to the derived class itself.

To model our different kinds of pricing strategies, we'll define a class named `Quote`, which will be the base class of our hierarchy. A `Quote` object will represent undiscounted books. From `Quote` we will inherit a second class, named `Bulk_quote`, to represent books that can be sold with a quantity discount.

These classes will have the following two member functions:

- `isbn()`, which will return the ISBN. This operation does not depend on the specifics of the inherited class(es); it will be defined only in class `Quote`.
- `net_price(size_t)`, which will return the price for purchasing a specified number of copies of a book. This operation is type specific; both `Quote` and `Bulk_quote` will define their own version of this function.

In C++, a base class distinguishes functions that are type dependent from those that it expects its derived classes to inherit without change. The base class defines as **virtual** those functions it expects its derived classes to define for themselves. Using this knowledge, we can start to write our `Quote` class:

```
class Quote {  
public:  
    std::string isbn() const;  
    virtual double net_price(std::size_t n) const;  
};
```

A derived class must specify the class(es) from which it intends to inherit. It does so in a **class derivation list**, which is a colon followed by a comma-separated list of base classes each of which may have an optional access specifier:

```
class Bulk_quote : public Quote { // Bulk_quote inherits from Quote
public:
    double net_price(std::size_t) const override;
};
```

Because `Bulk_quote` uses `public` in its derivation list, we can use objects of type `Bulk_quote` as if they were `Quote` objects.

A derived class must include in its own class body a declaration of all the virtual functions it intends to define for itself. A derived class may include the `virtual` keyword on these functions but is not required to do so. For reasons we'll explain in § 15.3 (p. 606), the new standard lets a derived class explicitly note that it intends a member function to **override** a virtual that it inherits. It does so by specifying `override` after its parameter list.

## Dynamic Binding

Through **dynamic binding**, we can use the same code to process objects of either type `Quote` or `Bulk_quote` interchangeably. For example, the following function prints the total price for purchasing the given number of copies of a given book:

```
// calculate and print the price for the given number of copies, applying any discounts
double print_total(ostream &os,
                   const Quote &item, size_t n)
{
    // depending on the type of the object bound to the item parameter
    // calls either Quote::net_price or Bulk_quote::net_price
    double ret = item.net_price(n);
    os << "ISBN: " << item.isbn() // calls Quote::isbn
    << " # sold: " << n << " total due: " << ret << endl;
    return ret;
}
```

This function is pretty simple—it prints the results of calling `isbn` and `net_price` on its parameter and returns the value calculated by the call to `net_price`.

Nevertheless, there are two interesting things about this function: For reasons we'll explain in § 15.2.3 (p. 601), because the `item` parameter is a reference to `Quote`, we can call this function on either a `Quote` object or a `Bulk_quote` object. And, for reasons we'll explain in § 15.2.1 (p. 594), because `net_price` is a virtual function, and because `print_total` calls `net_price` through a reference, the version of `net_price` that is run will depend on the type of the object that we pass to `print_total`:

```
// basic has type Quote; bulk has type Bulk_quote
print_total(cout, basic, 20); // calls Quote version of net_price
print_total(cout, bulk, 20); // calls Bulk_quote version of net_price
```

The first call passes a `Quote` object to `print_total`. When `print_total` calls `net_price`, the `Quote` version will be run. In the next call, the argument is a

`Bulk_quote`, so the `Bulk_quote` version of `net_price` (which applies a discount) will be run. Because the decision as to which version to run depends on the type of the argument, that decision can't be made until run time. Therefore, dynamic binding is sometimes known as **run-time binding**.



In C++, dynamic binding happens when a virtual function is called through a reference (or a pointer) to a base class.

## 15.2 Defining Base and Derived Classes

In many, but not all, ways base and derived classes are defined like other classes we have already seen. In this section, we'll cover the basic features used to define classes related by inheritance.



### 15.2.1 Defining a Base Class

We'll start by completing the definition of our `Quote` class:

```
class Quote {
public:
    Quote() = default; // = default see § 7.1.4 (p. 264)
    Quote(const std::string &book, double sales_price):
        bookNo(book), price(sales_price) { }
    std::string isbn() const { return bookNo; }
    // returns the total sales price for the specified number of items
    // derived classes will override and apply different discount algorithms
    virtual double net_price(std::size_t n) const
    { return n * price; }
    virtual ~Quote() = default; // dynamic binding for the destructor
private:
    std::string bookNo; // ISBN number of this item
protected:
    double price = 0.0; // normal, undiscounted price
};
```

The new parts in this class are the use of `virtual` on the `net_price` function and the destructor, and the `protected` access specifier. We'll explain virtual destructors in § 15.7.1 (p. 622), but for now it is worth noting that classes used as the root of an inheritance hierarchy almost always define a virtual destructor.



Base classes ordinarily should define a virtual destructor. Virtual destructors are needed even if they do no work.

## Member Functions and Inheritance

Derived classes inherit the members of their base class. However, a derived class needs to be able to provide its own definition for operations, such as `net_price`,

that are type dependent. In such cases, the derived class needs to **override** the definition it inherits from the base class, by providing its own definition.

In C++, a base class must distinguish the functions it expects its derived classes to override from those that it expects its derived classes to inherit without change. The base class defines as **virtual** those functions it expects its derived classes to override. When we call a virtual function *through a pointer or reference*, the call will be dynamically bound. Depending on the type of the object to which the reference or pointer is bound, the version in the base class or in one of its derived classes will be executed.

A base class specifies that a member function should be dynamically bound by preceding its declaration with the keyword **virtual**. Any nonstatic member function (§ 7.6, p. 300), other than a constructor, may be virtual. The **virtual** keyword appears only on the declaration inside the class and may not be used on a function definition that appears outside the class body. A function that is declared as **virtual** in the base class is implicitly **virtual** in the derived classes as well. We'll have more to say about virtual functions in § 15.3 (p. 603).

Member functions that are not declared as **virtual** are resolved at compile time, not run time. For the `isbn` member, this is exactly the behavior we want. The `isbn` function does not depend on the details of a derived type. It behaves identically when run on a `Quote` or `Bulk_quote` object. There will be only one version of the `isbn` function in our inheritance hierarchy. Thus, there is no question as to which function to run when we call `isbn()`.

## Access Control and Inheritance

A derived class inherits the members defined in its base class. However, the member functions in a derived class may not necessarily access the members that are inherited from the base class. Like any other code that uses the base class, a derived class may access the `public` members of its base class but may not access the `private` members. However, sometimes a base class has members that it wants to let its derived classes use while still prohibiting access to those same members by other users. We specify such members after a **protected** access specifier.

Our `Quote` class expects its derived classes to define their own `net_price` function. To do so, those classes need access to the `price` member. As a result, `Quote` defines that member as **protected**. Derived classes are expected to access `bookNo` in the same way as ordinary users—by calling the `isbn` function. Hence, the `bookNo` member is `private` and is inaccessible to classes that inherit from `Quote`. We'll have more to say about **protected** members in § 15.5 (p. 611).

### EXERCISES SECTION 15.2.1

**Exercise 15.1:** What is a virtual member?

**Exercise 15.2:** How does the **protected** access specifier differ from **private**?

**Exercise 15.3:** Define your own versions of the `Quote` class and the `print_total` function.



## 15.2.2 Defining a Derived Class

A derived class must specify from which class(es) it inherits. It does so in its **class derivation list**, which is a colon followed by a comma-separated list of names of previously defined classes. Each base class name may be preceded by an optional access specifier, which is one of `public`, `protected`, or `private`.

A derived class must declare each inherited member function it intends to override. Therefore, our `Bulk_quote` class must include a `net_price` member:

```
class Bulk_quote : public Quote { // Bulk_quote inherits from Quote
    Bulk_quote() = default;
    Bulk_quote(const std::string&, double, std::size_t, double);
    // overrides the base version in order to implement the bulk purchase discount policy
    double net_price(std::size_t) const override;
private:
    std::size_t min_qty = 0; // minimum purchase for the discount to apply
    double discount = 0.0;   // fractional discount to apply
};
```

Our `Bulk_quote` class inherits the `isbn` function and the `bookNo` and `price` data members of its `Quote` base class. It defines its own version of `net_price` and has two additional data members, `min_qty` and `discount`. These members specify the minimum quantity and the discount to apply once that number of copies are purchased.

We'll have more to say about the access specifier used in a derivation list in § 15.5 (p. 612). For now, what's useful to know is that the access specifier determines whether users of a derived class are allowed to know that the derived class inherits from its base class.

When the derivation is `public`, the `public` members of the base class become part of the interface of the derived class as well. In addition, we can bind an object of a publicly derived type to a pointer or reference to the base type. Because we used `public` in the derivation list, the interface to `Bulk_quote` implicitly contains the `isbn` function, and we may use a `Bulk_quote` object where a pointer or reference to `Quote` is expected.

Most classes inherit directly from only one base class. This form of inheritance, known as "single inheritance," forms the topic of this chapter. § 18.3 (p. 802) will cover classes that have derivation lists with more than one base class.

## Virtual Functions in the Derived Class

Derived classes frequently, but not always, override the virtual functions that they inherit. If a derived class does not override a virtual from its base, then, like any other member, the derived class inherits the version defined in its base class.

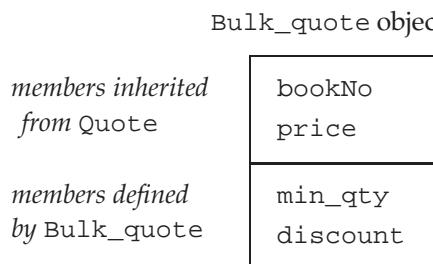
A derived class may include the `virtual` keyword on the functions it overrides, but it is not required to do so. For reasons we'll explain in § 15.3 (p. 606), the new standard lets a derived class explicitly note that it intends a member function to override a virtual that it inherits. It does so by specifying `override` after the parameter list, or after the `const` or reference qualifier(s) if the member is a `const` (§ 7.1.2, p. 258) or reference (§ 13.6.3, p. 546) function.

## Derived-Class Objects and the Derived-to-Base Conversion

A derived object contains multiple parts: a subobject containing the (nonstatic) members defined in the derived class itself, plus subobjects corresponding to each base class from which the derived class inherits. Thus, a `Bulk_quote` object will contain four data elements: the `bookNo` and `price` data members that it inherits from `Quote`, and the `min_qty` and `discount` members, which are defined by `Bulk_quote`.

Although the standard does not specify how derived objects are laid out in memory, we can think of a `Bulk_quote` object as consisting of two parts as represented in Figure 15.1.

**Figure 15.1: Conceptual Structure of a `Bulk_quote` Object**



**The base and derived parts of an object are not guaranteed to be stored contiguously.**

Figure 15.1 is a conceptual, not physical, representation of how classes work.

Because a derived object contains subparts corresponding to its base class(es), we can use an object of a derived type *as if* it were an object of its base type(s). In particular, we can bind a base-class reference or pointer to the base-class part of a derived object.

```

Quote item;           // object of base type
Bulk_quote bulk;     // object of derived type
Quote *p = &item;    // p points to a Quote object
p = &bulk;          // p points to the Quote part of bulk
Quote &r = bulk;    // r bound to the Quote part of bulk
    
```

This conversion is often referred to as the **derived-to-base** conversion. As with any other conversion, the compiler will apply the derived-to-base conversion implicitly (§ 4.11, p. 159).

The fact that the derived-to-base conversion is implicit means that we can use an object of derived type or a reference to a derived type when a reference to the base type is required. Similarly, we can use a pointer to a derived type where a pointer to the base type is required.



The fact that a derived object contains subobjects for its base classes is key to how inheritance works.

## Derived-Class Constructors

Although a derived object contains members that it inherits from its base, it cannot directly initialize those members. Like any other code that creates an object of the base-class type, a derived class must use a base-class constructor to initialize its base-class part.



Each class controls how its members are initialized.

The base-class part of an object is initialized, along with the data members of the derived class, during the initialization phase of the constructor (§ 7.5.1, p. 288). Analogously to how we initialize a member, a derived-class constructor uses its constructor initializer list to pass arguments to a base-class constructor. For example, the `Bulk_quote` constructor with four parameters:

```
Bulk_quote(const std::string& book, double p,
           std::size_t qty, double disc) :
    Quote(book, p), min_qty(qty), discount(disc) { }
    // as before
};
```

passes its first two parameters (representing the ISBN and price) to the `Quote` constructor. That `Quote` constructor initializes the `Bulk_quote`'s base-class part (i.e., the `bookNo` and `price` members). When the (empty) `Quote` constructor body completes, the base-class part of the object being constructed will have been initialized. Next the direct members, `min_qty` and `discount`, are initialized. Finally, the (empty) function body of the `Bulk_quote` constructor is run.

As with a data member, unless we say otherwise, the base part of a derived object is default initialized. To use a different base-class constructor, we provide a constructor initializer using the name of the base class, followed (as usual) by a parenthesized list of arguments. Those arguments are used to select which base-class constructor to use to initialize the base-class part of the derived object.



The base class is initialized first, and then the members of the derived class are initialized in the order in which they are declared in the class.

## Using Members of the Base Class from the Derived Class

A derived class may access the `public` and `protected` members of its base class:

```
// if the specified number of items are purchased, use the discounted price
double Bulk_quote::net_price(size_t cnt) const
{
    if (cnt >= min_qty)
        return cnt * (1 - discount) * price;
    else
        return cnt * price;
}
```

This function generates a discounted price: If the given quantity is more than `min_qty`, we apply the `discount` (which was stored as a fraction) to the `price`.

We'll have more to say about scope in § 15.6 (p. 617), but for now it's worth knowing that the scope of a derived class is nested inside the scope of its base class. As a result, there is no distinction between how a member of the derived class uses members defined in its own class (e.g., `min_qty` and `discount`) and how it uses members defined in its base (e.g., `price`).

### KEY CONCEPT: RESPECTING THE BASE-CLASS INTERFACE

It is essential to understand that each class defines its own interface. Interactions with an object of a class-type should use the interface of that class, even if that object is the base-class part of a derived object.

As a result, derived-class constructors may not directly initialize the members of its base class. The constructor body of a derived constructor can assign values to its `public` or `protected` base-class members. Although it *can* assign to those members, it generally *should not* do so. Like any other user of the base class, a derived class should respect the interface of its base class by using a constructor to initialize its inherited members.

## Inheritance and `static` Members

If a base class defines a `static` member (§ 7.6, p. 300), there is only one such member defined for the entire hierarchy. Regardless of the number of classes derived from a base class, there exists a single instance of each `static` member.

```
class Base {
public:
    static void statmem();
};

class Derived : public Base {
    void f(const Derived&);
};
```

`static` members obey normal access control. If the member is `private` in the base class, then derived classes have no access to it. Assuming the member is `accessible`, we can use a `static` member through either the base or derived:

```
void Derived::f(const Derived &derived_obj)
{
    Base::statmem();      // ok: Base defines statmem
    Derived::statmem();   // ok: Derived inherits statmem
    // ok: derived objects can be used to access static from base
    derived_obj.statmem(); // accessed through a Derived object
    statmem();            // accessed through this object
}
```

## Declarations of Derived Classes

A derived class is declared like any other class (§ 7.3.3, p. 278). The declaration contains the class name but does not include its derivation list:

```
class Bulk_quote : public Quote; // error: derivation list can't appear here
class Bulk_quote; // ok: right way to declare a derived class
```

The purpose of a declaration is to make known that a name exists and what kind of entity it denotes, for example, a class, function, or variable. The derivation list, and all other details of the definition, must appear together in the class body.

## Classes Used as a Base Class

A class must be defined, not just declared, before we can use it as a base class:

```
class Quote; // declared but not defined
// error: Quote must be defined
class Bulk_quote : public Quote { ... };
```

The reason for this restriction should be easy to see: Each derived class contains, and may use, the members it inherits from its base class. To use those members, the derived class must know what they are. One implication of this rule is that it is impossible to derive a class from itself.

A base class can itself be a derived class:

```
class Base { /* ... */ };
class D1: public Base { /* ... */ };
class D2: public D1 { /* ... */ };
```

In this hierarchy, `Base` is a **direct base** to `D1` and an **indirect base** to `D2`. A direct base class is named in the derivation list. An indirect base is one that a derived class inherits through its direct base class.

Each class inherits all the members of its direct base class. The most derived class inherits the members of its direct base. The members in the direct base include those it inherits from its base class, and so on up the inheritance chain. Effectively, the most derived object contains a subobject for its direct base and for each of its indirect bases.

## Preventing Inheritance

Sometimes we define a class that we don't want others to inherit from. Or we might define a class for which we don't want to think about whether it is appropriate as a base class. Under the new standard, we can prevent a class from being used as a base by following the class name with `final`:

C++  
11

```
class NoDerived final { /* */ }; // NoDerived can't be a base class
class Base { /* */ };
// Last is final; we cannot inherit from Last
class Last final : Base { /* */ }; // Last can't be a base class
class Bad : NoDerived { /* */ }; // error: NoDerived is final
class Bad2 : Last { /* */ }; // error: Last is final
```

## EXERCISES SECTION 15.2.2

**Exercise 15.4:** Which of the following declarations, if any, are incorrect? Explain why.

- ```
class Base { ... };
(a) class Derived : public Derived { ... };
(b) class Derived : private Base { ... };
(c) class Derived : public Base;
```

**Exercise 15.5:** Define your own version of the `Bulk_quote` class.

**Exercise 15.6:** Test your `print_total` function from the exercises in § 15.2.1 (p. 595) by passing both `Quote` and `Bulk_quote` objects to that function.

**Exercise 15.7:** Define a class that implements a limited discount strategy, which applies a discount to books purchased up to a given limit. If the number of copies exceeds that limit, the normal price applies to those purchased beyond the limit.

### 15.2.3 Conversions and Inheritance



Understanding conversions between base and derived classes is essential to understanding how object-oriented programming works in C++.

Ordinarily, we can bind a reference or a pointer only to an object that has the same type as the corresponding reference or pointer (§ 2.3.1, p. 51, and § 2.3.2, p. 52) or to a type that involves an acceptable `const` conversion (§ 4.11.2, p. 162). Classes related by inheritance are an important exception: We can bind a pointer or reference to a base-class type to an object of a type derived from that base class. For example, we can use a `Quote&` to refer to a `Bulk_quote` object, and we can assign the address of a `Bulk_quote` object to a `Quote*`.

The fact that we can bind a reference (or pointer) to a base-class type to a derived object has a crucially important implication: When we use a reference (or pointer) to a base-class type, we don't know the actual type of the object to which the pointer or reference is bound. That object can be an object of the base class or it can be an object of a derived class.



Like built-in pointers, the smart pointer classes (§ 12.1, p. 450) support the derived-to-base conversion—we can store a pointer to a derived object in a smart pointer to the base type.

### Static Type and Dynamic Type



When we use types related by inheritance, we often need to distinguish between the **static type** of a variable or other expression and the **dynamic type** of the object that expression represents. The static type of an expression is always known at compile time—it is the type with which a variable is declared or that an expression yields. The dynamic type is the type of the object in memory that the variable or expression represents. The dynamic type may not be known until run time.

For example, when `print_total` calls `net_price` (§ 15.1, p. 593):

```
double ret = item.net_price(n);
```

we know that the static type of `item` is `Quote&`. The dynamic type depends on the type of the argument to which `item` is bound. That type cannot be known until a call is executed at run time. If we pass a `Bulk_quote` object to `print_total`, then the static type of `item` will differ from its dynamic type. As we've seen, the static type of `item` is `Quote&`, but in this case the dynamic type is `Bulk_quote`.

The dynamic type of an expression that is neither a reference nor a pointer is always the same as that expression's static type. For example, a variable of type `Quote` is always a `Quote` object; there is nothing we can do that will change the type of the object to which that variable corresponds.



It is crucial to understand that the static type of a pointer or reference to a base class may differ from its dynamic type.

## There Is No Implicit Conversion from Base to Derived ...

The conversion from derived to base exists because every derived object contains a base-class part to which a pointer or reference of the base-class type can be bound. There is no similar guarantee for base-class objects. A base-class object can exist either as an independent object or as part of a derived object. A base object that is not part of a derived object has only the members defined by the base class; it doesn't have the members defined by the derived class.

Because a base object might or might not be part of a derived object, there is no automatic conversion from the base class to its derived class(s):

```
Quote base;
Bulk_quote* bulkP = &base; // error: can't convert base to derived
Bulk_quote& bulkRef = base; // error: can't convert base to derived
```

If these assignments were legal, we might attempt to use `bulkP` or `bulkRef` to use members that do not exist in `base`.

What is sometimes a bit surprising is that we cannot convert from base to derived even when a base pointer or reference is bound to a derived object:

```
Bulk_quote bulk;
Quote *itemP = &bulk;           // ok: dynamic type is Bulk_quote
Bulk_quote *bulkP = itemP;     // error: can't convert base to derived
```

The compiler has no way to know (at compile time) that a specific conversion will be safe at run time. The compiler looks only at the static types of the pointer or reference to determine whether a conversion is legal. If the base class has one or more virtual functions, we can use a `dynamic_cast` (which we'll cover in § 19.2.1 (p. 825)) to request a conversion that is checked at run time. Alternatively, in those cases when we *know* that the conversion from base to derived is safe, we can use a `static_cast` (§ 4.11.3, p. 162) to override the compiler.

## ...and No Conversion between Objects

The automatic derived-to-base conversion applies only for conversions to a reference or pointer type. There is no such conversion from a derived-class type to the base-class type. Nevertheless, it is often possible to convert an object of a derived class to its base-class type. However, such conversions may not behave as we might want.

Remember that when we initialize or assign an object of a class type, we are actually calling a function. When we initialize, we're calling a constructor (§ 13.1.1, p. 496, and § 13.6.2, p. 534); when we assign, we're calling an assignment operator (§ 13.1.2, p. 500, and § 13.6.2, p. 536). These members normally have a parameter that is a reference to the `const` version of the class type.

Because these members take references, the derived-to-base conversion lets us pass a derived object to a base-class copy/move operation. These operations are not virtual. When we pass a derived object to a base-class constructor, the constructor that is run is defined in the base class. That constructor knows *only* about the members of the base class itself. Similarly, if we assign a derived object to a base object, the assignment operator that is run is the one defined in the base class. That operator also knows *only* about the members of the base class itself.

For example, our bookstore classes use the synthesized versions of copy and assignment (§ 13.1.1, p. 497, and § 13.1.2, p. 500). We'll have more to say about copy control and inheritance in § 15.7.2 (p. 623), but for now what's useful to know is that the synthesized versions memberwise copy or assign the data members of the class the same way as for any other class:

```
Bulk_quote bulk; // object of derived type
Quote item(bulk); // uses the Quote::Quote(const Quote&) constructor
item = bulk; // calls Quote::operator=(const Quote&)
```

When `item` is constructed, the `Quote` copy constructor is run. That constructor knows only about the `bookNo` and `price` members. It copies those members from the `Quote` part of `bulk` and *ignores* the members that are part of the `Bulk_quote` portion of `bulk`. Similarly for the assignment of `bulk` to `item`; only the `Quote` part of `bulk` is assigned to `item`.

Because the `Bulk_quote` part is ignored, we say that the `Bulk_quote` portion of `bulk` is **sliced down**.



When we initialize or assign an object of a base type from an object of a derived type, only the base-class part of the derived object is copied, moved, or assigned. The derived part of the object is ignored.

## 15.3 Virtual Functions



As we've seen, in C++ dynamic binding happens when a virtual member function is called through a reference or a pointer to a base-class type (§ 15.1, p. 593). Because we don't know which version of a function is called until run time, virtual functions must *always* be defined. Ordinarily, if we do not use a function, we don't

### EXERCISES SECTION 15.2.3

**Exercise 15.8:** Define static type and dynamic type.

**Exercise 15.9:** When is it possible for an expression's static type to differ from its dynamic type? Give three examples in which the static and dynamic type differ.

**Exercise 15.10:** Recalling the discussion from § 8.1 (p. 311), explain how the program on page 317 that passed an `ifstream` to the `Sales_data` read function works.

### KEY CONCEPT: CONVERSIONS AMONG TYPES RELATED BY INHERITANCE

There are three things that are important to understand about conversions among classes related by inheritance:

- The conversion from derived to base applies only to pointer or reference types.
- There is no implicit conversion from the base-class type to the derived type.
- Like any member, the derived-to-base conversion may be inaccessible due to access controls. We'll cover accessibility in § 15.5 (p. 613).

Although the automatic conversion applies only to pointers and references, most classes in an inheritance hierarchy (implicitly or explicitly) define the copy-control members (Chapter 13). As a result, we can often copy, move, or assign an object of derived type to a base-type object. However, copying, moving, or assigning a derived-type object to a base-type object copies, moves, or assigns *only* the members in the base-class part of the object.

need to supply a definition for that function (§ 6.1.2, p. 206). However, we must define every virtual function, regardless of whether it is used, because the compiler has no way to determine whether a virtual function is used.

### Calls to Virtual Functions May Be Resolved at Run Time

When a virtual function is called through a reference or pointer, the compiler generates code to *decide at run time* which function to call. The function that is called is the one that corresponds to the dynamic type of the object bound to that pointer or reference.

As an example, consider our `print_total` function from § 15.1 (p. 593). That function calls `net_price` on its parameter named `item`, which has type `Quote&`. Because `item` is a reference, and because `net_price` is virtual, the version of `net_price` that is called depends at run time on the actual (dynamic) type of the argument bound to `item`:

```
Quote base("0-201-82470-1", 50);
print_total(cout, base, 10);      // calls Quote::net_price
Bulk_quote derived("0-201-82470-1", 50, 5, .19);
print_total(cout, derived, 10); // calls Bulk_quote::net_price
```

In the first call, `item` is bound to an object of type `Quote`. As a result, when

`print_total` calls `net_price`, the version defined by `Quote` is run. In the second call, `item` is bound to a `Bulk_quote` object. In this call, `print_total` calls the `Bulk_quote` version of `net_price`.

It is crucial to understand that dynamic binding happens only when a virtual function is called through a pointer or a reference.

```
base = derived;           // copies the Quote part of derived into base
base.net_price(20);      // calls Quote::net_price
```

When we call a virtual function on an expression that has a plain—nonreference and nonpointer—type, that call is bound at compile time. For example, when we call `net_price` on `base`, there is no question as to which version of `net_price` to run. We can change the value (i.e., the contents) of the object that `base` represents, but there is no way to change the type of that object. Hence, this call is resolved, at compile time, to the `Quote` version of `net_price`.

### KEY CONCEPT: POLYMORPHISM IN C++

The key idea behind OOP is polymorphism. Polymorphism is derived from a Greek word meaning “many forms.” We speak of types related by inheritance as polymorphic types, because we can use the “many forms” of these types while ignoring the differences among them. The fact that the static and dynamic types of references and pointers can differ is the cornerstone of how C++ supports polymorphism.

When we call a function defined in a base class through a reference or pointer to the base class, we do not know the type of the object on which that member is executed. The object can be a base-class object or an object of a derived class. If the function is virtual, then the decision as to which function to run is delayed until run time. The version of the virtual function that is run is the one defined by the type of the object to which the reference is bound or to which the pointer points.

On the other hand, calls to nonvirtual functions are bound at compile time. Similarly, calls to any function (virtual or not) on an object are also bound at compile time. The type of an object is fixed and unvarying—there is nothing we can do to make the dynamic type of an object differ from its static type. Therefore, calls made on an object are bound at compile time to the version defined by the type of the object.



Virtuals are resolved at run time *only* if the call is made through a reference or pointer. Only in these cases is it possible for an object's dynamic type to differ from its static type.

## Virtual Functions in a Derived Class

When a derived class overrides a virtual function, it may, but is not required to, repeat the `virtual` keyword. Once a function is declared as `virtual`, it remains `virtual` in all the derived classes.

A derived-class function that overrides an inherited virtual function must have exactly the same parameter type(s) as the base-class function that it overrides.

With one exception, the return type of a virtual in the derived class also must match the return type of the function from the base class. The exception applies to

virtuals that return a reference (or pointer) to types that are themselves related by inheritance. That is, if D is derived from B, then a base class virtual can return a B\* and the version in the derived can return a D\*. However, such return types require that the derived-to-base conversion from D to B is accessible. § 15.5 (p. 613) covers how to determine whether a base class is accessible. We'll see an example of this kind of virtual function in § 15.8.1 (p. 633).



A function that is `virtual` in a base class is implicitly `virtual` in its derived classes. When a derived class overrides a virtual, the parameters in the base and derived classes must match exactly.

## The `final` and `override` Specifiers

As we'll see in § 15.6 (p. 620), it is legal for a derived class to define a function with the same name as a virtual in its base class but with a different parameter list. The compiler considers such a function to be independent from the base-class function. In such cases, the derived version does not override the version in the base class. In practice, such declarations often are a mistake—the class author intended to override a virtual from the base class but made a mistake in specifying the parameter list.

C++  
11

Finding such bugs can be surprisingly hard. Under the new standard we can specify `override` on a virtual function in a derived class. Doing so makes our intention clear and (more importantly) enlists the compiler in finding such problems for us. The compiler will reject a program if a function marked `override` does not override an existing virtual function:

```
struct B {
    virtual void f1(int) const;
    virtual void f2();
    void f3();
};

struct D1 : B {
    void f1(int) const override; // ok: f1 matches f1 in the base
    void f2(int) override; // error: B has no f2(int) function
    void f3() override; // error: f3 not virtual
    void f4() override; // error: B doesn't have a function named f4
};
```

In D1, the `override` specifier on `f1` is fine; both the base and derived versions of `f1` are `const` members that take an `int` and return `void`. The version of `f1` in D1 properly overrides the virtual that it inherits from B.

The declaration of `f2` in D1 does not match the declaration of `f2` in B—the version defined in B takes no arguments and the one defined in D1 takes an `int`. Because the declarations don't match, `f2` in D1 doesn't override `f2` from B; it is a new function that happens to have the same name. Because we said we intended this declaration to be an `override` and it isn't, the compiler will generate an error.

Because only a virtual function can be overridden, the compiler will also reject `f3` in D1. That function is not virtual in B, so there is no function to override.

Similarly `f4` is in error because `B` doesn't even have a function named `f4`.

We can also designate a function as `final`. Any attempt to override a function that has been defined as `final` will be flagged as an error:

```
struct D2 : B {  
    // inherits f2() and f3() from B and overrides f1(int)  
    void f1(int) const final; // subsequent classes can't override f1(int)  
};  
  
struct D3 : D2 {  
    void f2(); // ok: overrides f2 inherited from the indirect base, B  
    void f1(int) const; // error: D2 declared f2 as final  
};
```

`final` and `override` specifiers appear after the parameter list (including any `const` or reference qualifiers) and after a trailing return (§ 6.3.3, p. 229).

## Virtual Functions and Default Arguments

Like any other function, a virtual function can have default arguments (§ 6.5.1, p. 236). If a call uses a default argument, the value that is used is the one defined by the static type through which the function is called.

That is, when a call is made through a reference or pointer to base, the default argument(s) will be those defined in the base class. The base-class arguments will be used even when the derived version of the function is run. In this case, the derived function will be passed the default arguments defined for the base-class version of the function. If the derived function relies on being passed different arguments, the program will not execute as expected.



Virtual functions that have default arguments should use the same argument values in the base and derived classes.

## Circumventing the Virtual Mechanism

In some cases, we want to prevent dynamic binding of a call to a virtual function; we want to force the call to use a particular version of that virtual. We can use the scope operator to do so. For example, this code:

```
// calls the version from the base class regardless of the dynamic type of baseP  
double undiscounted = baseP->Quote::net_price(42);
```

calls the `Quote` version of `net_price` regardless of the type of the object to which `baseP` actually points. This call will be resolved at compile time.



Ordinarily, only code inside member functions (or friends) should need to use the scope operator to circumvent the virtual mechanism.

Why might we wish to circumvent the virtual mechanism? The most common reason is when a derived-class virtual function calls the version from the base class. In such cases, the base-class version might do work common to all types in the hierarchy. The versions defined in the derived classes would do whatever additional work is particular to their own type.



If a derived virtual function that intended to call its base-class version omits the scope operator, the call will be resolved at run time as a call to the derived version itself, resulting in an infinite recursion.

## EXERCISES SECTION 15.3

**Exercise 15.11:** Add a virtual debug function to your `Quote` class hierarchy that displays the data members of the respective classes.

**Exercise 15.12:** Is it ever useful to declare a member function as both `override` and `final`? Why or why not?

**Exercise 15.13:** Given the following classes, explain each `print` function:

```
class base {
public:
    string name() { return basename; }
    virtual void print(ostream &os) { os << basename; }
private:
    string basename;
};

class derived : public base {
public:
    void print(ostream &os) { print(os); os << " " << i; }
private:
    int i;
};
```

If there is a problem in this code, how would you fix it?

**Exercise 15.14:** Given the classes from the previous exercise and the following objects, determine which function is called at run time:

```
base bobj;      base *bp1 = &bobj;      base &br1 = bobj;
derived dobj;  base *bp2 = &dobj;      base &br2 = dobj;
(a) bobj.print(); (b) dobj.print(); (c) bp1->name();
(d) bp2->name(); (e) br1.print(); (f) br2.print();
```

## 15.4 Abstract Base Classes

Imagine that we want to extend our bookstore classes to support several discount strategies. In addition to a bulk discount, we might offer a discount for purchases up to a certain quantity and then charge the full price thereafter. Or we might offer a discount for purchases above a certain limit but not for purchases up to that limit.

Each of these discount strategies is the same in that it requires a quantity and a discount amount. We might support these differing strategies by defining a new class named `Disc_quote` to store the quantity and the discount amount. Classes, such as `Bulk_item`, that represent a specific discount strategy will inherit from

`Disc_quote`. Each of the derived classes will implement its discount strategy by defining its own version of `net_price`.

Before we can define our `Disc_Quote` class, we have to decide what to do about `net_price`. Our `Disc_quote` class doesn't correspond to any particular discount strategy; there is no meaning to ascribe to `net_price` for this class.

We could define `Disc_quote` without its own version of `net_price`. In this case, `Disc_quote` would inherit `net_price` from `Quote`.

However, this design would make it possible for our users to write nonsensical code. A user could create an object of type `Disc_quote` by supplying a quantity and a discount rate. Passing that `Disc_quote` object to a function such as `print_total` would use the `Quote` version of `net_price`. The calculated price would not include the discount that was supplied when the object was created. That state of affairs makes no sense.

## Pure Virtual Functions

Thinking about the question in this detail reveals that our problem is not just that we don't know how to define `net_price`. In practice, we'd like to prevent users from creating `Disc_quote` objects at all. This class represents the general concept of a discounted book, not a concrete discount strategy.

We can enforce this design intent—and make it clear that there is no meaning for `net_price`—by defining `net_price` as a **pure virtual** function. Unlike ordinary virtuals, a pure virtual function does not have to be defined. We specify that a virtual function is a pure virtual by writing `= 0` in place of a function body (i.e., just before the semicolon that ends the declaration). The `= 0` may appear only on the declaration of a virtual function in the class body:

```
// class to hold the discount rate and quantity
// derived classes will implement pricing strategies using these data
class Disc_quote : public Quote {
public:
    Disc_quote() = default;
    Disc_quote(const std::string& book, double price,
               std::size_t qty, double disc):
        Quote(book, price),
        quantity(qty), discount(disc) { }
    double net_price(std::size_t) const = 0;
protected:
    std::size_t quantity = 0; // purchase size for the discount to apply
    double discount = 0.0;    // fractional discount to apply
};
```

Like our earlier `Bulk_item` class, `Disc_quote` defines a default constructor and a constructor that takes four parameters. Although we cannot define objects of this type directly, constructors in classes derived from `Disc_quote` will use the `Disc_quote` constructors to construct the `Disc_quote` part of their objects. The constructor that has four parameters passes its first two to the `Quote` constructor and directly initializes its own members, `discount` and `quantity`. The default constructor default initializes those members.

It is worth noting that we can provide a definition for a pure virtual. However, the function body must be defined outside the class. That is, we cannot provide a function body inside the class for a function that is = 0.

## Classes with Pure Virtuals Are Abstract Base Classes

A class containing (or inheriting without overriding) a pure virtual function is an **abstract base class**. An abstract base class defines an interface for subsequent classes to override. We cannot (directly) create objects of a type that is an abstract base class. Because `Disc_quote` defines `net_price` as a pure virtual, we cannot define objects of type `Disc_quote`. We can define objects of classes that inherit from `Disc_quote`, so long as those classes override `net_price`:

```
// Disc_quote declares pure virtual functions, which Bulk_quote will override
Disc_quote discounted; // error: can't define a Disc_quote object
Bulk_quote bulk; // ok: Bulk_quote has no pure virtual functions
```

Classes that inherit from `Disc_quote` must define `net_price` or those classes will be abstract as well.



We may not create objects of a type that is an abstract base class.

## A Derived Class Constructor Initializes Its Direct Base Class Only

Now we can reimplement `Bulk_quote` to inherit from `Disc_quote` rather than inheriting directly from `Quote`:

```
// the discount kicks in when a specified number of copies of the same book are sold
// the discount is expressed as a fraction to use to reduce the normal price
class Bulk_quote : public Disc_quote {
public:
    Bulk_quote() = default;
    Bulk_quote(const std::string& book, double price,
               std::size_t qty, double disc):
        Disc_quote(book, price, qty, disc) { }
    // overrides the base version to implement the bulk purchase discount policy
    double net_price(std::size_t) const override;
};
```

This version of `Bulk_quote` has a direct base class, `Disc_quote`, and an indirect base class, `Quote`. Each `Bulk_quote` object has three subobjects: an (empty) `Bulk_quote` part, a `Disc_quote` subobject, and a `Quote` subobject.

As we've seen, each class controls the initialization of objects of its type. Therefore, even though `Bulk_quote` has no data members of its own, it provides the same four-argument constructor as in our original class. Our new constructor passes its arguments to the `Disc_quote` constructor. That constructor in turn runs the `Quote` constructor. The `Quote` constructor initializes the `bookNo` and `price` members of `bulk`. When the `Quote` constructor ends, the `Disc_quote` constructor runs and initializes the `quantity` and `discount` members. At this

point, the `Bulk_quote` constructor resumes. That constructor has no further initializations or any other work to do.

#### KEY CONCEPT: REFACTORING

Adding `Disc_quote` to the `Quote` hierarchy is an example of *refactoring*. Refactoring involves redesigning a class hierarchy to move operations and/or data from one class to another. Refactoring is common in object-oriented applications.

It is noteworthy that even though we changed the inheritance hierarchy, code that uses `Bulk_quote` or `Quote` would not need to change. However, when classes are refactored (or changed in any other way) we must recompile any code that uses those classes.

### EXERCISES SECTION 15.4

**Exercise 15.15:** Define your own versions of `Disc_quote` and `Bulk_quote`.

**Exercise 15.16:** Rewrite the class representing a limited discount strategy, which you wrote for the exercises in § 15.2.2 (p. 601), to inherit from `Disc_quote`.

**Exercise 15.17:** Try to define an object of type `Disc_quote` and see what errors you get from the compiler.

## 15.5 Access Control and Inheritance



Just as each class controls the initialization of its own members (§ 15.2.2, p. 598), each class also controls whether its members are **accessible** to a derived class.

### protected Members

As we've seen, a class uses `protected` for those members that it is willing to share with its derived classes but wants to protect from general access. The `protected` specifier can be thought of as a blend of `private` and `public`:

- Like `private`, `protected` members are inaccessible to users of the class.
- Like `public`, `protected` members are accessible to members and friends of classes derived from this class.

In addition, `protected` has another important property:

- A derived class member or friend may access the `protected` members of the base class *only* through a derived object. The derived class has no special access to the `protected` members of base-class objects.

To understand this last rule, consider the following example:

```

class Base {
protected:
    int prot_mem;      // protected member
};

class Sneaky : public Base {
    friend void clobber(Sneaky&); // can access Sneaky::prot_mem
    friend void clobber(Base&);   // can't access Base::prot_mem
    int j;                // j is private by default
};

// ok: clobber can access the private and protected members in Sneaky objects
void clobber(Sneaky &s) { s.j = s.prot_mem = 0; }

// error: clobber can't access the protected members in Base
void clobber(Base &b) { b.prot_mem = 0; }

```

If derived classes (and friends) could access protected members in a base-class object, then our second version of `clobber` (that takes a `Base&`) would be legal. That function is not a friend of `Base`, yet it would be allowed to change an object of type `Base`; we could circumvent the protection provided by `protected` for any class simply by defining a new class along the lines of `Sneaky`.

To prevent such usage, members and friends of a derived class can access the `protected` members *only* in base-class objects that are embedded inside a derived type object; they have no special access to ordinary objects of the base type.

## **public, private, and protected Inheritance**

Access to a member that a class inherits is controlled by a combination of the access specifier for that member in the base class, and the access specifier in the derivation list of the derived class. As an example, consider the following hierarchy:

```

class Base {
public:
    void pub_mem();      // public member
protected:
    int prot_mem;      // protected member
private:
    char priv_mem;    // private member
};

struct Pub_Derv : public Base {
    // ok: derived classes can access protected members
    int f() { return prot_mem; }
    // error: private members are inaccessible to derived classes
    char g() { return priv_mem; }
};

struct Priv_Derv : private Base {
    // private derivation doesn't affect access in the derived class
    int f1() const { return prot_mem; }
};

```

The derivation access specifier has no effect on whether members (and friends) of a derived class may access the members of its own direct base class. Access to the

members of a base class is controlled by the access specifiers in the base class itself. Both `Pub_Derv` and `Priv_Derv` may access the protected member `prot_mem`. Neither may access the private member `priv_mem`.

The purpose of the derivation access specifier is to control the access that *users* of the derived class—including other classes derived from the derived class—have to the members inherited from `Base`:

```
Pub_Derv d1; // members inherited from Base are public
Priv_Derv d2; // members inherited from Base are private
d1.pub_mem(); // ok: pub_mem is public in the derived class
d2.pub_mem(); // error: pub_mem is private in the derived class
```

Both `Pub_Derv` and `Priv_Derv` inherit the `pub_mem` function. When the inheritance is `public`, members retain their access specification. Thus, `d1` can call `pub_mem`. In `Priv_Derv`, the members of `Base` are `private`; users of that class may not call `pub_mem`.

The derivation access specifier used by a derived class also controls access from classes that inherit from that derived class:

```
struct Derived_from_Public : public Pub_Derv {
    // ok: Base::prot_mem remains protected in Pub_Derv
    int use_base() { return prot_mem; }
};

struct Derived_from_Private : public Priv_Derv {
    // error: Base::prot_mem is private in Priv_Derv
    int use_base() { return prot_mem; }
};
```

Classes derived from `Pub_Derv` may access `prot_mem` from `Base` because that member remains a protected member in `Pub_Derv`. In contrast, classes derived from `Priv_Derv` have no such access. To them, all the members that `Priv_Derv` inherited from `Base` are `private`.

Had we defined another class, say, `Prot_Derv`, that used `protected` inheritance, the `public` members of `Base` would be `protected` members in that class. Users of `Prot_Derv` would have no access to `pub_mem`, but the members and friends of `Prot_Derv` could access that inherited member.

## Accessibility of Derived-to-Base Conversion

Whether the derived-to-base conversion (§ 15.2.2, p. 597) is accessible depends on which code is trying to use the conversion and may depend on the access specifier used in the derived class' derivation. Assuming `D` inherits from `B`:

- User code may use the derived-to-base conversion *only* if `D` inherits publicly from `B`. User code may not use the conversion if `D` inherits from `B` using either `protected` or `private`.
- Member functions and friends of `D` can use the conversion to `B` regardless of how `D` inherits from `B`. The derived-to-base conversion to a direct base class is always accessible to members and friends of a derived class.



- Member functions and friends of classes derived from D may use the derived-to-base conversion if D inherits from B using either `public` or `protected`. Such code may not use the conversion if D inherits privately from B.



For any given point in your code, if a `public` member of the base class would be accessible, then the derived-to-base conversion is also accessible, and not otherwise.

### KEY CONCEPT: CLASS DESIGN AND PROTECTED MEMBERS

In the absence of inheritance, we can think of a class as having two different kinds of users: ordinary users and implementors. Ordinary users write code that uses objects of the class type; such code can access only the `public` (interface) members of the class. Implementors write the code contained in the members and friends of the class. The members and friends of the class can access both the `public` and `private` (implementation) sections.

Under inheritance, there is a third kind of user, namely, derived classes. A base class makes `protected` those parts of its implementation that it is willing to let its derived classes use. The `protected` members remain inaccessible to ordinary user code; `private` members remain inaccessible to derived classes and their friends.

Like any other class, a class that is used as a base class makes its interface members `public`. A class that is used as a base class may divide its implementation into those members that are accessible to derived classes and those that remain accessible only to the base class and its friends. An implementation member should be `protected` if it provides an operation or data that a derived class will need to use in its own implementation. Otherwise, implementation members should be `private`.

## Friendship and Inheritance

Just as friendship is not transitive (§ 7.3.4, p. 279), friendship is also not inherited. Friends of the base have no special access to members of its derived classes, and friends of a derived class have no special access to the base class:

```
class Base {
    // added friend declaration; other members as before
    friend class Pal; // Pal has no access to classes derived from Base
};
class Pal {
public:
    int f(Base b) { return b.prot_mem; } // ok: Pal is a friend of Base
    int f2(Sneaky s) { return s.j; } // error: Pal not friend of Sneaky
    // access to a base class is controlled by the base class, even inside a derived object
    int f3(Sneaky s) { return s.prot_mem; } // ok: Pal is a friend
};
```

The fact that `f3` is legal may seem surprising, but it follows directly from the notion that each class controls access to its own members. `Pal` is a friend of `Base`, so

Pal can access the members of Base objects. That access includes access to Base objects that are embedded in an object of a type derived from Base.

When a class makes another class a friend, it is only that class to which friendship is granted. The base classes of, and classes derived from, the friend have no special access to the befriending class:

```
// D2 has no access to protected or private members in Base
class D2 : public Pal {
public:
    int mem(Base b)
        { return b.prot_mem; } // error: friendship doesn't inherit
};
```



Friendship is not inherited; each class controls access to its members.

## Exempting Individual Members

Sometimes we need to change the access level of a name that a derived class inherits. We can do so by providing a using declaration (§ 3.1, p. 82):

```
class Base {
public:
    std::size_t size() const { return n; }
protected:
    std::size_t n;
};

class Derived : private Base { // note: private inheritance
public:
    // maintain access levels for members related to the size of the object
    using Base::size;
protected:
    using Base::n;
};
```

Because Derived uses private inheritance, the inherited members, size and n, are (by default) private members of Derived. The using declarations adjust the accessibility of these members. Users of Derived can access the size member, and classes subsequently derived from Derived can access n.

A using declaration inside a class can name any accessible (e.g., not private) member of a direct or indirect base class. Access to a name specified in a using declaration depends on the access specifier preceding the using declaration. That is, if a using declaration appears in a private part of the class, that name is accessible to members and friends only. If the declaration is in a public section, the name is available to all users of the class. If the declaration is in a protected section, the name is accessible to the members, friends, and derived classes.



A derived class may provide a using declaration only for names it is permitted to access.

## Default Inheritance Protection Levels

In § 7.2 (p. 268) we saw that classes defined with the `struct` and `class` keywords have different default access specifiers. Similarly, the default derivation specifier depends on which keyword is used to define a derived class. By default, a derived class defined with the `class` keyword has `private` inheritance; a derived class defined with `struct` has `public` inheritance:

```
class Base { /* ... */ };
struct D1 : Base { /* ... */ }; // public inheritance by default
class D2 : Base { /* ... */ }; // private inheritance by default
```

It is a common misconception to think that there are deeper differences between classes defined using the `struct` keyword and those defined using `class`. The only differences are the default access specifier for members and the default derivation access specifier. There are no other distinctions.



A privately derived class should specify `private` explicitly rather than rely on the default. Being explicit makes it clear that private inheritance is intended and not an oversight.

### EXERCISES SECTION 15.5

**Exercise 15.18:** Given the classes from page 612 and page 613, and assuming each object has the type specified in the comments, determine which of these assignments are legal. Explain why those that are illegal aren't allowed:

```
Base *p = &d1; // d1 has type Pub_Derv
p = &d2; // d2 has type Priv_Derv
p = &d3; // d3 has type Prot_Derv
p = &dd1; // dd1 has type Derived_from_Public
p = &dd2; // dd2 has type Derived_from_Private
p = &dd3; // dd3 has type Derived_from_Protected
```

**Exercise 15.19:** Assume that each of the classes from page 612 and page 613 has a member function of the form:

```
void memfcn(Base &b) { b = *this; }
```

For each class, determine whether this function would be legal.

**Exercise 15.20:** Write code to test your answers to the previous two exercises.

**Exercise 15.21:** Choose one of the following general abstractions containing a family of types (or choose one of your own). Organize the types into an inheritance hierarchy:

- (a) Graphical file formats (such as gif, tiff, jpeg, bmp)
- (b) Geometric primitives (such as box, circle, sphere, cone)
- (c) C++ language types (such as class, function, member function)

**Exercise 15.22:** For the class you chose in the previous exercise, identify some of the likely virtual functions as well as `public` and `protected` members.

## 15.6 Class Scope under Inheritance



Each class defines its own scope (§ 7.4, p. 282) within which its members are defined. Under inheritance, the scope of a derived class is nested (§ 2.2.4, p. 48) inside the scope of its base classes. If a name is unresolved within the scope of the derived class, the enclosing base-class scopes are searched for a definition of that name.

The fact that the scope of a derived class nests inside the scope of its base classes can be surprising. After all, the base and derived classes are defined in separate parts of our program's text. However, it is this hierarchical nesting of class scopes that allows the members of a derived class to use members of its base class as if those members were part of the derived class. For example, when we write

```
Bulk_quote bulk;
cout << bulk.isbn();
```

the use of the name `isbn` is resolved as follows:

- Because we called `isbn` on an object of type `Bulk_quote`, the search starts in the `Bulk_quote` class. The name `isbn` is not found in that class.
- Because `Bulk_quote` is derived from `Disc_quote`, the `Disc_quote` class is searched next. The name is still not found.
- Because `Disc_quote` is derived from `Quote`, the `Quote` class is searched next. The name `isbn` is found in that class; the use of `isbn` is resolved to the `isbn` in `Quote`.

### Name Lookup Happens at Compile Time

The static type (§ 15.2.3, p. 601) of an object, reference, or pointer determines which members of that object are visible. Even when the static and dynamic types might differ (as can happen when a reference or pointer to a base class is used), the static type determines what members can be used. As an example, we might add a member to the `Disc_quote` class that returns a `pair` (§ 11.2.3, p. 426) holding the minimum (or maximum) quantity and the discounted price:

```
class Disc_quote : public Quote {
public:
    std::pair<size_t, double> discount_policy() const
        { return {quantity, discount}; }
    // other members as before
};
```

We can use `discount_policy` only through an object, pointer, or reference of type `Disc_quote` or of a class derived from `Disc_quote`:

```
Bulk_quote bulk;
Bulk_quote *bulkP = &bulk; // static and dynamic types are the same
Quote *itemP = &bulk; // static and dynamic types differ
bulkP->discount_policy(); // ok: bulkP has type Bulk_quote*
itemP->discount_policy(); // error: itemP has type Quote*
```

Even though `bulk` has a member named `discount_policy`, that member is not visible through `itemP`. The type of `itemP` is a pointer to `Quote`, which means that the search for `discount_policy` starts in class `Quote`. The `Quote` class has no member named `discount_policy`, so we cannot call that member on an object, reference, or pointer of type `Quote`.

## Name Collisions and Inheritance

Like any other scope, a derived class can reuse a name defined in one of its direct or indirect base classes. As usual, names defined in an inner scope (e.g., a derived class) hide uses of that name in the outer scope (e.g., a base class) (§ 2.2.4, p. 48):

```
struct Base {
    Base(): mem(0) { }
protected:
    int mem;
};

struct Derived : Base {
    Derived(int i): mem(i) { } // initializes Derived::mem to i
                           // Base::mem is default initialized
    int get_mem() { return mem; } // returns Derived::mem
protected:
    int mem; // hides mem in the base
};
```

The reference to `mem` inside `get_mem` is resolved to the name inside `Derived`. Were we to write

```
Derived d(42);
cout << d.get_mem() << endl; // prints 42
```

then the output would be 42.



A derived-class member with the same name as a member of the base class hides direct use of the base-class member.

## Using the Scope Operator to Use Hidden Members

We can use a hidden base-class member by using the scope operator:

```
struct Derived : Base {
    int get_base_mem() { return Base::mem; }
    // ...
};
```

The scope operator overrides the normal lookup and directs the compiler to look for `mem` starting in the scope of class `Base`. If we ran the code above with this version of `Derived`, the result of `d.get_mem()` would be 0.



Aside from overriding inherited virtual functions, a derived class usually should not reuse names defined in its base class.

**KEY CONCEPT: NAME LOOKUP AND INHERITANCE**

Understanding how function calls are resolved is crucial to understanding inheritance in C++. Given the call `p->mem()` (or `obj.mem()`), the following four steps happen:

- First determine the static type of `p` (or `obj`). Because we're calling a member, that type must be a class type.
- Look for `mem` in the class that corresponds to the static type of `p` (or `obj`). If `mem` is not found, look in the direct base class and continue up the chain of classes until `mem` is found or the last class is searched. If `mem` is not found in the class or its enclosing base classes, then the call will not compile.
- Once `mem` is found, do normal type checking (§ 6.1, p. 203) to see if this call is legal given the definition that was found.
- Assuming the call is legal, the compiler generates code, which varies depending on whether the call is virtual or not:
  - If `mem` is virtual and the call is made through a reference or pointer, then the compiler generates code to determine at run time which version to run based on the dynamic type of the object.
  - Otherwise, if the function is nonvirtual, or if the call is on an object (not a reference or pointer), the compiler generates a normal function call.

## As Usual, Name Lookup Happens before Type Checking

As we've seen, functions declared in an inner scope do not overload functions declared in an outer scope (§ 6.4.1, p. 234). As a result, functions defined in a derived class do *not* overload members defined in its base class(es). As in any other scope, if a member in a derived class (i.e., in an inner scope) has the same name as a base-class member (i.e., a name defined in an outer scope), then the derived member hides the base-class member within the scope of the derived class. The base member is hidden even if the functions have different parameter lists:

```
struct Base {
    int memfcn();
};

struct Derived : Base {
    int memfcn(int); // hides memfcn in the base
};

Derived d; Base b;

b.memfcn(); // calls Base::memfcn
d.memfcn(10); // calls Derived::memfcn
d.memfcn(); // error: memfcn with no arguments is hidden
d.Base::memfcn(); // ok: calls Base::memfcn
```

The declaration of `memfcn` in `Derived` hides the declaration of `memfcn` in `Base`. Not surprisingly, the first call through `b`, which is a `Base` object, calls the version in the base class. Similarly, the second call (through `d`) calls the one from `Derived`. What can be surprising is that the third call, `d.memfcn()`, is illegal.

To resolve this call, the compiler looks for the name `memfcn` in `Derived`. That class defines a member named `memfcn` and the search stops. Once the name is found, the compiler looks no further. The version of `memfcn` in `Derived` expects an `int` argument. This call provides no such argument; it is in error.



## Virtual Functions and Scope

We can now understand why virtual functions must have the same parameter list in the base and derived classes (§ 15.3, p. 605). If the base and derived members took arguments that differed from one another, there would be no way to call the derived version through a reference or pointer to the base class. For example:

```
class Base {
public:
    virtual int fcn();
};

class D1 : public Base {
public:
    // hides fcn in the base; this fcn is not virtual
    // D1 inherits the definition of Base::fcn()
    int fcn(int);           // parameter list differs from fcn in Base
    virtual void f2();      // new virtual function that does not exist in Base
};

class D2 : public D1 {
public:
    int fcn(int);          // nonvirtual function hides D1::fcn(int)
    int fcn();              // overrides virtual fcn from Base
    void f2();              // overrides virtual f2 from D1
};
```

The `fcn` function in `D1` does not override the virtual `fcn` from `Base` because they have different parameter lists. Instead, it *hides* `fcn` from the base. Effectively, `D1` has two functions named `fcn`: `D1` inherits a virtual named `fcn` from `Base` and defines its own, nonvirtual member named `fcn` that takes an `int` parameter.

## Calling a Hidden Virtual through the Base Class

Given the classes above, let's look at several different ways to call these functions:

```
Base bobj; D1 d1obj; D2 d2obj;
Base *bp1 = &bobj, *bp2 = &d1obj, *bp3 = &d2obj;
bp1->fcn(); // virtual call, will call Base::fcn at run time
bp2->fcn(); // virtual call, will call Base::fcn at run time
bp3->fcn(); // virtual call, will call D2::fcn at run time
D1 *d1p = &d1obj; D2 *d2p = &d2obj;
bp2->f2(); // error: Base has no member named f2
d1p->f2(); // virtual call, will call D1::f2() at run time
d2p->f2(); // virtual call, will call D2::f2() at run time
```

The first three calls are all made through pointers to the base class. Because `fcn` is virtual, the compiler generates code to decide at run time which version to call.

That decision will be based on the actual type of the object to which the pointer is bound. In the case of `bp2`, the underlying object is a `D1`. That class did not override the `fcn` function that takes no arguments. Thus, the call through `bp2` is resolved (at run time) to the version defined in `Base`.

The next three calls are made through pointers with differing types. Each pointer points to one of the types in this hierarchy. The first call is illegal because there is no `f2()` in class `Base`. The fact that the pointer happens to point to a derived object is irrelevant.

For completeness, let's look at calls to the nonvirtual function `fcn(int)`:

```
Base *p1 = &d2obj; D1 *p2 = &d2obj; D2 *p3 = &d2obj;
p1->fcn(42); // error: Base has no version of fcn that takes an int
p2->fcn(42); // statically bound, calls D1::fcn(int)
p3->fcn(42); // statically bound, calls D2::fcn(int)
```

In each call the pointer happens to point to an object of type `D2`. However, the dynamic type doesn't matter when we call a nonvirtual function. The version that is called depends only on the static type of the pointer.

## Overriding Overloaded Functions

As with any other function, a member function (virtual or otherwise) can be overloaded. A derived class can override zero or more instances of the overloaded functions it inherits. If a derived class wants to make all the overloaded versions available through its type, then it must override all of them or none of them.

Sometimes a class needs to override some, but not all, of the functions in an overloaded set. It would be tedious in such cases to have to override every base-class version in order to override the ones that the class needs to specialize.

Instead of overriding every base-class version that it inherits, a derived class can provide a `using` declaration (§ 15.5, p. 615) for the overloaded member. A `using` declaration specifies only a name; it may not specify a parameter list. Thus, a `using` declaration for a base-class member function adds all the overloaded instances of that function to the scope of the derived class. Having brought all the names into its scope, the derived class needs to define only those functions that truly depend on its type. It can use the inherited definitions for the others.

The normal rules for a `using` declaration inside a class apply to names of overloaded functions (§ 15.5, p. 615); every overloaded instance of the function in the base class must be accessible to the derived class. The access to the overloaded versions that are not otherwise redefined by the derived class will be the access in effect at the point of the `using` declaration.

### EXERCISES SECTION 15.6

**Exercise 15.23:** Assuming class `D1` on page 620 had intended to override its inherited `fcn` function, how would you fix that class? Assuming you fixed the class so that `fcn` matched the definition in `Base`, how would the calls in that section be resolved?

## 15.7 Constructors and Copy Control

Like any other class, a class in an inheritance hierarchy controls what happens when objects of its type are created, copied, moved, assigned, or destroyed. As for any other class, if a class (base or derived) does not itself define one of the copy-control operations, the compiler will synthesize that operation. Also, as usual, the synthesized version of any of these members might be a deleted function.



### 15.7.1 Virtual Destructors

The primary direct impact that inheritance has on copy control for a base class is that a base class generally should define a virtual destructor (§ 15.2.1, p. 594). The destructor needs to be virtual to allow objects in the inheritance hierarchy to be dynamically allocated.

Recall that the destructor is run when we delete a pointer to a dynamically allocated object (§ 13.1.3, p. 502). If that pointer points to a type in an inheritance hierarchy, it is possible that the static type of the pointer might differ from the dynamic type of the object being destroyed (§ 15.2.2, p. 597). For example, if we delete a pointer of type `Quote*`, that pointer might point at a `Bulk_quote` object. If the pointer points at a `Bulk_quote`, the compiler has to know that it should run the `Bulk_quote` destructor. As with any other function, we arrange to run the proper destructor by defining the destructor as virtual in the base class:

```
class Quote {
public:
    // virtual destructor needed if a base pointer pointing to a derived object is deleted
    virtual ~Quote() = default; // dynamic binding for the destructor
};
```

Like any other virtual, the virtual nature of the destructor is inherited. Thus, classes derived from `Quote` have virtual destructors, whether they use the synthesized destructor or define their own version. So long as the base class destructor is virtual, when we delete a pointer to base, the correct destructor will be run:

```
Quote *itemP = new Quote; // same static and dynamic type
delete itemP;           // destructor for Quote called
itemP = new Bulk_quote; // static and dynamic types differ
delete itemP;           // destructor for Bulk_quote called
```



**WARNING** Executing `delete` on a pointer to base that points to a derived object has undefined behavior if the base's destructor is not virtual.

Destructors for base classes are an important exception to the rule of thumb that if a class needs a destructor, it also needs copy and assignment (§ 13.1.4, p. 504). A base class almost always needs a destructor, so that it can make the destructor virtual. If a base class has an empty destructor in order to make it virtual, then the fact that the class has a destructor does not indicate that the assignment operator or copy constructor is also needed.

## Virtual Destructors Turn Off Synthesized Move

The fact that a base class needs a virtual destructor has an important indirect impact on the definition of base and derived classes: If a class defines a destructor—even if it uses `= default` to use the synthesized version—the compiler will not synthesize a move operation for that class (§ 13.6.2, p. 537).

### EXERCISES SECTION 15.7.1

**Exercise 15.24:** What kinds of classes need a virtual destructor? What operations must a virtual destructor perform?

## 15.7.2 Synthesized Copy Control and Inheritance



The synthesized copy-control members in a base or a derived class execute like any other synthesized constructor, assignment operator, or destructor: They memberwise initialize, assign, or destroy the members of the class itself. In addition, these synthesized members initialize, assign, or destroy the direct base part of an object by using the corresponding operation from the base class. For example,

- The synthesized `Bulk_quote` default constructor runs the `Disc_Quote` default constructor, which in turn runs the `Quote` default constructor.
- The `Quote` default constructor initializes the `bookNo` member to the empty string and uses the in-class initializer to initialize `price` to zero.
- When the `Quote` constructor finishes, the `Disc_Quote` constructor continues, which uses the in-class initializers to initialize `qty` and `discount`.
- When the `Disc_quote` constructor finishes, the `Bulk_quote` constructor continues but has no other work to do.

Similarly, the synthesized `Bulk_quote` copy constructor uses the (synthesized) `Disc_quote` copy constructor, which uses the (synthesized) `Quote` copy constructor. The `Quote` copy constructor copies the `bookNo` and `price` members; and the `Disc_Quote` copy constructor copies the `qty` and `discount` members.

It is worth noting that it doesn't matter whether the base-class member is itself synthesized (as is the case in our `Quote` hierarchy) or has a user-provided definition. All that matters is that the corresponding member is accessible (§ 15.5, p. 611) and that it is not a deleted function.

Each of our `Quote` classes use the synthesized destructor. The derived classes do so implicitly, whereas the `Quote` class does so explicitly by defining its (virtual) destructor as `= default`. The synthesized destructor is (as usual) empty and its implicit destruction part destroys the members of the class (§ 13.1.3, p. 501). In addition to destroying its own members, the destruction phase of a destructor in a derived class also destroys its direct base. That destructor in turn invokes the destructor for its own direct base, if any. And, so on up to the root of the hierarchy.

As we've seen, `Quote` does not have synthesized move operations because it defines a destructor. The (synthesized) copy operations will be used whenever we move a `Quote` object (§ 13.6.2, p. 540). As we're about to see, the fact that `Quote` does not have move operations means that its derived classes don't either.

## Base Classes and Deleted Copy Control in the Derived

C++  
11

The synthesized default constructor, or any of the copy-control members of either a base or a derived class, may be defined as deleted for the same reasons as in any other class (§ 13.1.6, p. 508, and § 13.6.2, p. 537). In addition, the way in which a base class is defined can cause a derived-class member to be defined as deleted:

- If the default constructor, copy constructor, copy-assignment operator, or destructor in the base class is deleted or inaccessible (§ 15.5, p. 612), then the corresponding member in the derived class is defined as deleted, because the compiler can't use the base-class member to construct, assign, or destroy the base-class part of the object.
- If the base class has an inaccessible or deleted destructor, then the synthesized default and copy constructors in the derived classes are defined as deleted, because there is no way to destroy the base part of the derived object.
- As usual, the compiler will not synthesize a deleted move operation. If we use `= default` to request a move operation, it will be a deleted function in the derived if the corresponding operation in the base is deleted or inaccessible, because the base class part cannot be moved. The move constructor will also be deleted if the base class destructor is deleted or inaccessible.

As an example, this base class, `B`,

```
class B {
public:
    B();
    B(const B&) = delete;
    // other members, not including a move constructor
};
class D : public B {
    // no constructors
};
D d;      // ok: D's synthesized default constructor uses B's default constructor
D d2(d); // error: D's synthesized copy constructor is deleted
D d3(std::move(d)); // error: implicitly uses D's deleted copy constructor
```

has an accessible default constructor and an explicitly deleted copy constructor. Because the copy constructor is defined, the compiler will not synthesize a move constructor for class `B` (§ 13.6.2, p. 537). As a result, we can neither move nor copy objects of type `B`. If a class derived from `B` wanted to allow its objects to be copied or moved, that derived class would have to define its own versions of these constructors. Of course, that class would have to decide how to copy or move the members in its base-class part. In practice, if a base class does not have a default, copy, or move constructor, then its derived classes usually don't either.

## Move Operations and Inheritance

As we've seen, most base classes define a virtual destructor. As a result, by default, base classes generally do not get synthesized move operations. Moreover, by default, classes derived from a base class that doesn't have move operations don't get synthesized move operations either.

Because lack of a move operation in a base class suppresses synthesized move for its derived classes, base classes ordinarily should define the move operations if it is sensible to do so. Our `Quote` class can use the synthesized versions. However, `Quote` must define these members explicitly. Once it defines its move operations, it must also explicitly define the copy versions as well (§ 13.6.2, p. 539):

```
class Quote {  
public:  
    Quote() = default;           // memberwise default initialize  
    Quote(const Quote&) = default; // memberwise copy  
    Quote(Quote&&) = default;     // memberwise copy  
    Quote& operator=(const Quote&) = default; // copy assign  
    Quote& operator=(Quote&&) = default;      // move assign  
    virtual ~Quote() = default;  
    // other members as before  
};
```

Now, `Quote` objects will be memberwise copied, moved, assigned, and destroyed. Moreover, classes derived from `Quote` will automatically obtain synthesized move operations as well, unless they have members that otherwise preclude move.

### EXERCISES SECTION 15.7.2

**Exercise 15.25:** Why did we define a default constructor for `Disc_quote`? What effect, if any, would removing that constructor have on the behavior of `Bulk_quote`?

### 15.7.3 Derived-Class Copy-Control Members



As we saw in § 15.2.2 (p. 598), the initialization phase of a derived-class constructor initializes the base-class part(s) of a derived object as well as initializing its own members. As a result, the copy and move constructors for a derived class must copy/move the members of its base part as well as the members in the derived. Similarly, a derived-class assignment operator must assign the members in the base part of the derived object.

Unlike the constructors and assignment operators, the destructor is responsible only for destroying the resources allocated by the derived class. Recall that the members of an object are implicitly destroyed (§ 13.1.3, p. 502). Similarly, the base-class part of a derived object is destroyed automatically.



When a derived class defines a copy or move operation, that operation is responsible for copying or moving the entire object, including base-class members.



## Defining a Derived Copy or Move Constructor

When we define a copy or move constructor (§ 13.1.1, p. 496, and § 13.6.2, p. 534) for a derived class, we ordinarily use the corresponding base-class constructor to initialize the base part of the object:

```
class Base { /* ... */ };
class D: public Base {
public:
    // by default, the base class default constructor initializes the base part of an object
    // to use the copy or move constructor, we must explicitly call that
    // constructor in the constructor initializer list
    D(const D& d): Base(d)           // copy the base members
        /* initializers for members of D */ { /* ... */ }
    D(D&& d): Base(std::move(d)) // move the base members
        /* initializers for members of D */ { /* ... */ }
};
```

The initializer `Base(d)` passes a `D` object to a base-class constructor. Although in principle, `Base` could have a constructor that has a parameter of type `D`, in practice, that is very unlikely. Instead, `Base(d)` will (ordinarily) match the `Base` copy constructor. The `D` object, `d`, will be bound to the `Base&` parameter in that constructor. The `Base` copy constructor will copy the base part of `d` into the object that is being created. Had the initializer for the base class been omitted,

```
// probably incorrect definition of the D copy constructor
// base-class part is default initialized, not copied
D(const D& d) /* member initializers, but no base-class initializer */ 
    { /* ... */ }
```

the `Base` default constructor would be used to initialize the base part of a `D` object. Assuming `D`'s constructor copies the derived members from `d`, this newly constructed object would be oddly configured: Its `Base` members would hold default values, while its `D` members would be copies of the data from another object.



By default, the base-class default constructor initializes the base-class part of a derived object. If we want copy (or move) the base-class part, we must explicitly use the copy (or move) constructor for the base class in the derived's constructor initializer list.

## Derived-Class Assignment Operator

Like the copy and move constructors, a derived-class assignment operator (§ 13.1.2, p. 500, and § 13.6.2, p. 536), must assign its base part explicitly:

```

// Base::operator=(const Base&) is not invoked automatically
D &D::operator=(const D &rhs)
{
    Base::operator=(rhs); // assigns the base part
    // assign the members in the derived class, as usual,
    // handling self-assignment and freeing existing resources as appropriate
    return *this;
}

```

This operator starts by explicitly calling the base-class assignment operator to assign the members of the base part of the derived object. The base-class operator will (presumably) correctly handle self-assignment and, if appropriate, will free the old value in the base part of the left-hand operand and assign the new values from rhs. Once that operator finishes, we continue doing whatever is needed to assign the members in the derived class.

It is worth noting that a derived constructor or assignment operator can use its corresponding base class operation regardless of whether the base defined its own version of that operator or uses the synthesized version. For example, the call to `Base::operator=` executes the copy-assignment operator in class `Base`. It is immaterial whether that operator is defined explicitly by the `Base` class or is synthesized by the compiler.

## Derived-Class Destructor

Recall that the data members of an object are implicitly destroyed after the destructor body completes (§ 13.1.3, p. 502). Similarly, the base-class parts of an object are also implicitly destroyed. As a result, unlike the constructors and assignment operators, a derived destructor is responsible only for destroying the resources allocated by the derived class:

```

class D: public Base {
public:
    // Base::~Base invoked automatically
    ~D() { /* do what it takes to clean up derived members */ }
};

```

Objects are destroyed in the opposite order from which they are constructed: The derived destructor is run first, and then the base-class destructors are invoked, back up through the inheritance hierarchy.

## Calls to Virtuals in Constructors and Destructors

As we've seen, the base-class part of a derived object is constructed first. While the base-class constructor is executing, the derived part of the object is uninitialized. Similarly, derived objects are destroyed in reverse order, so that when a base class destructor runs, the derived part has already been destroyed. As a result, while these base-class members are executing, the object is incomplete.

To accommodate this incompleteness, the compiler treats the object as if its type changes during construction or destruction. That is, while an object is being constructed it is treated as if it has the same class as the constructor; calls to virtual

functions will be bound as if the object has the same type as the constructor itself. Similarly, for destructors. This binding applies to virtuals called directly or that are called indirectly from a function that the constructor (or destructor) calls.

To understand this behavior, consider what would happen if the derived-class version of a virtual was called from a base-class constructor. This virtual probably accesses members of the derived object. After all, if the virtual didn't need to use members of the derived object, the derived class probably could use the version in its base class. However, those members are uninitialized while a base constructor is running. If such access were allowed, the program would probably crash.



If a constructor or destructor calls a virtual, the version that is run is the one corresponding to the type of the constructor or destructor itself.

### EXERCISES SECTION 15.7.3

**Exercise 15.26:** Define the `Quote` and `Bulk_quote` copy-control members to do the same job as the synthesized versions. Give them and the other constructors print statements that identify which function is running. Write programs using these classes and predict what objects will be created and destroyed. Compare your predictions with the output and continue experimenting until your predictions are reliably correct.

#### 15.7.4 Inherited Constructors

C++  
11

Under the new standard, a derived class can reuse the constructors defined by its direct base class. Although, as we'll see, such constructors are not inherited in the normal sense of that term, it is nonetheless common to refer to such constructors as "inherited." For the same reasons that a class may initialize only its direct base class, a class may inherit constructors only from its direct base. A class cannot inherit the default, copy, and move constructors. If the derived class does not directly define these constructors, the compiler synthesizes them as usual.

A derived class inherits its base-class constructors by providing a `using` declaration that names its (direct) base class. As an example, we can redefine our `Bulk_quote` class (§ 15.4, p. 610) to inherit its constructors from `Disc_quote`:

```
class Bulk_quote : public Disc_quote {
public:
    using Disc_quote::Disc_quote; // inherit Disc_quote's constructors
    double net_price(std::size_t) const;
};
```

Ordinarily, a `using` declaration only makes a name visible in the current scope. When applied to a constructor, a `using` declaration causes the compiler to generate code. The compiler generates a derived constructor corresponding to each constructor in the base. That is, for each constructor in the base class, the compiler generates a constructor in the derived class that has the same parameter list.

These compiler-generated constructors have the form

```
derived(parms) : base(args) { }
```

where *derived* is the name of the derived class, *base* is the name of the base class, *parms* is the parameter list of the constructor, and *args* pass the parameters from the derived constructor to the base constructor. In our *Bulk\_quote* class, the inherited constructor would be equivalent to

```
Bulk_quote(const std::string& book, double price,
           std::size_t qty, double disc):
    Disc_quote(book, price, qty, disc) { }
```

If the derived class has any data members of its own, those members are default initialized (§ 7.1.4, p. 266).

## Characteristics of an Inherited Constructor

Unlike using declarations for ordinary members, a constructor using declaration does not change the access level of the inherited constructor(s). For example, regardless of where the using declaration appears, a private constructor in the base is a private constructor in the derived; similarly for protected and public constructors.

Moreover, a using declaration can't specify explicit or constexpr. If a constructor in the base is explicit (§ 7.5.4, p. 296) or constexpr (§ 7.5.6, p. 299), the inherited constructor has the same property.

If a base-class constructor has default arguments (§ 6.5.1, p. 236), those arguments are not inherited. Instead, the derived class gets multiple inherited constructors in which each parameter with a default argument is successively omitted. For example, if the base has a constructor with two parameters, the second of which has a default, the derived class will obtain two constructors: one with both parameters (and no default argument) and a second constructor with a single parameter corresponding to the left-most, non-defaulted parameter in the base class.

If a base class has several constructors, then with two exceptions, the derived class inherits each of the constructors from its base class. The first exception is that a derived class can inherit some constructors and define its own versions of other constructors. If the derived class defines a constructor with the same parameters as a constructor in the base, then that constructor is not inherited. The one defined in the derived class is used in place of the inherited constructor.

The second exception is that the default, copy, and move constructors are not inherited. These constructors are synthesized using the normal rules. An inherited constructor is not treated as a user-defined constructor. Therefore, a class that contains only inherited constructors will have a synthesized default constructor.

### EXERCISES SECTION 15.7.4

**Exercise 15.27:** Redefine your *Bulk\_quote* class to inherit its constructors.



## 15.8 Containers and Inheritance

When we use a container to store objects from an inheritance hierarchy, we generally must store those objects indirectly. We cannot put objects of types related by inheritance directly into a container, because there is no way to define a container that holds elements of differing types.

As an example, assume we want to define a `vector` to hold several books that a customer wants to buy. It should be easy to see that we can't use a `vector` that holds `Bulk_quote` objects. We can't convert `Quote` objects to `Bulk_quote` (§ 15.2.3, p. 602), so we wouldn't be able to put `Quote` objects into that `vector`.

It may be somewhat less obvious that we also can't use a `vector` that holds objects of type `Quote`. In this case, we can put `Bulk_quote` objects into the container. However, those objects would no longer be `Bulk_quote` objects:

```
vector<Quote> basket;
basket.push_back(Quote("0-201-82470-1", 50));
// ok, but copies only the Quote part of the object into basket
basket.push_back(Bulk_quote("0-201-54848-8", 50, 10, .25));
// calls version defined by Quote, prints 750, i.e., 15 * $50
cout << basket.back().net_price(15) << endl;
```

The elements in `basket` are `Quote` objects. When we add a `Bulk_quote` object to the `vector` its derived part is ignored (§ 15.2.3, p. 603).



Because derived objects are “sliced down” when assigned to a base-type object, containers and types related by inheritance do not mix well.

### Put (Smart) Pointers, Not Objects, in Containers

When we need a container that holds objects related by inheritance, we typically define the container to hold pointers (preferably smart pointers (§ 12.1, p. 450)) to the base class. As usual, the dynamic type of the object to which those pointers point might be the base-class type or a type derived from that base:

```
vector<shared_ptr<Quote>> basket;
basket.push_back(make_shared<Quote>("0-201-82470-1", 50));
basket.push_back(
    make_shared<Bulk_quote>("0-201-54848-8", 50, 10, .25));
// calls the version defined by Quote; prints 562.5, i.e., 15 * $50 less the discount
cout << basket.back()->net_price(15) << endl;
```

Because `basket` holds `shared_ptr`s, we must dereference the value returned by `basket.back()` to get the object on which to run `net_price`. We do so by using `->` in the call to `net_price`. As usual, the version of `net_price` that is called depends on the dynamic type of the object to which that pointer points.

It is worth noting that we defined `basket` as `shared_ptr<Quote>`, yet in the second `push_back` we passed a `shared_ptr` to a `Bulk_quote` object. Just as we can convert an ordinary pointer to a derived type to a pointer to an base-class type (§ 15.2.2, p. 597), we can also convert a smart pointer to a derived type to a

smart pointer to an base-class type. Thus, `make_shared<Bulk_quote>` returns a `shared_ptr<Bulk_quote>` object, which is converted to `shared_ptr<Quote>` when we call `push_back`. As a result, despite appearances, all of the elements of `basket` have the same type.

## EXERCISES SECTION 15.8

**Exercise 15.28:** Define a vector to hold `Quote` objects but put `Bulk_quote` objects into that vector. Compute the total `net_price` of all the elements in the vector.

**Exercise 15.29:** Repeat your program, but this time store `shared_ptrs` to objects of type `Quote`. Explain any discrepancy in the sum generated by the this version and the previous program. If there is no discrepancy, explain why there isn't one.

### 15.8.1 Writing a Basket Class



One of the ironies of object-oriented programming in C++ is that we cannot use objects directly to support it. Instead, we must use pointers and references. Because pointers impose complexity on our programs, we often define auxiliary classes to help manage that complexity. We'll start by defining a class to represent a basket:

```
class Basket {
public:
    // Basket uses synthesized default constructor and copy-control members
    void add_item(const std::shared_ptr<Quote> &sale)
        { items.insert(sale); }
    // prints the total price for each book and the overall total for all items in the basket
    double total_receipt(std::ostream&) const;
private:
    // function to compare shared_ptrs needed by the multiset member
    static bool compare(const std::shared_ptr<Quote> &lhs,
                        const std::shared_ptr<Quote> &rhs)
    { return lhs->isbn() < rhs->isbn(); }
    // multiset to hold multiple quotes, ordered by the compare member
    std::multiset<std::shared_ptr<Quote>, decltype(compare)*>
        items{compare};
};
```

Our class uses a `multiset` (§ 11.2.1, p. 423) to hold the transactions, so that we can store multiple transactions for the same book, and so that all the transactions for a given book will be kept together (§ 11.2.2, p. 424).

The elements in our `multiset` are `shared_ptrs` and there is no less-than operator for `shared_ptr`. As a result, we must provide our own comparison operation to order the elements (§ 11.2.2, p. 425). Here, we define a `private static` member, named `compare`, that compares the `isbns` of the objects to which the `shared_ptrs` point. We initialize our `multiset` to use this comparison function through an in-class initializer (§ 7.3.1, p. 274):

```
// multiset to hold multiple quotes, ordered by the compare member
std::multiset<std::shared_ptr<Quote>, decltype(compare)*>
    items{compare};
```

This declaration can be hard to read, but reading from left to right, we see that we are defining a multiset of shared\_ptrs to Quote objects. The multiset will use a function with the same type as our compare member to order the elements. The multiset member is named items, and we're initializing items to use our compare function.

## Defining the Members of Basket

The Basket class defines only two operations. We defined the add\_item member inside the class. That member takes a shared\_ptr to a dynamically allocated Quote and puts that shared\_ptr into the multiset. The second member, total\_receipt, prints an itemized bill for the contents of the basket and returns the price for all the items in the basket:

```
double Basket::total_receipt(ostream &os) const
{
    double sum = 0.0;      // holds the running total

    // iter refers to the first element in a batch of elements with the same ISBN
    // upper_bound returns an iterator to the element just past the end of that batch
    for (auto iter = items.cbegin();
         iter != items.cend();
         iter = items.upper_bound(*iter)) {
        // we know there's at least one element with this key in the Basket
        // print the line item for this book
        sum += print_total(os, **iter, items.count(*iter));
    }
    os << "Total Sale: " << sum << endl; // print the final overall total
    return sum;
}
```

Our for loop starts by defining and initializing iter to refer to the first element in the multiset. The condition checks whether iter is equal to items.cend(). If so, we've processed all the purchases and we drop out of the for. Otherwise, we process the next book.

The interesting bit is the “increment” expression in the for. Rather than the usual loop that reads each element, we advance iter to refer to the next key. We skip over all the elements that match the current key by calling upper\_bound (§ 11.3.5, p. 438). The call to upper\_bound returns the iterator that refers to the element just past the last one with the same key as in iter. The iterator we get back denotes either the end of the set or the next book.

Inside the for loop, we call print\_total (§ 15.1, p. 593) to print the details for each book in the basket:

```
sum += print_total(os, **iter, items.count(*iter));
```

The arguments to print\_total are an ostream on which to write, a Quote object to process, and a count. When we dereference iter, we get a shared\_ptr

that points to the object we want to print. To get that object, we must dereference that `shared_ptr`. Thus, `**iter` is a `Quote` object (or an object of a type derived from `Quote`). We use the `multiset::count` member (§ 11.3.5, p. 436) to determine how many elements in the `multiset` have the same key (i.e., the same ISBN).

As we've seen, `print_total` makes a virtual call to `net_price`, so the resulting price depends on the dynamic type of `**iter`. The `print_total` function prints the total for the given book and returns the total price that it calculated. We add that result into `sum`, which we print after we complete the `for` loop.

## Hiding the Pointers

Users of `Basket` still have to deal with dynamic memory, because `add_item` takes a `shared_ptr`. As a result, users have to write code such as

```
Basket bsk;
bsk.add_item(make_shared<Quote>("123", 45));
bsk.add_item(make_shared<Bulk_quote>("345", 45, 3, .15));
```

Our next step will be to redefine `add_item` so that it takes a `Quote` object instead of a `shared_ptr`. This new version of `add_item` will handle the memory allocation so that our users no longer need to do so. We'll define two versions, one that will copy its given object and the other that will move from it (§ 13.6.3, p. 544):

```
void add_item(const Quote& sale); // copy the given object
void add_item(Quote&& sale); // move the given object
```

The only problem is that `add_item` doesn't know what type to allocate. When it does its memory allocation, `add_item` will copy (or move) its `sale` parameter. Somewhere there will be a new expression such as:

```
new Quote(sale)
```

Unfortunately, this expression won't do the right thing: `new` allocates an object of the type we request. This expression allocates an object of type `Quote` and copies the `Quote` portion of `sale`. However, `sale` might refer to a `Bulk_quote` object, in which case, that object will be sliced down.

## Simulating Virtual Copy

We'll solve this problem by giving our `Quote` classes a virtual member that allocates a copy of itself.

```
class Quote {
public:
    // virtual function to return a dynamically allocated copy of itself
    // these members use reference qualifiers; see § 13.6.3 (p. 546)
    virtual Quote* clone() const & {return new Quote(*this);}
    virtual Quote* clone() &&
        {return new Quote(std::move(*this));}
    // other members as before
};
```

```

class Bulk_quote : public Quote {
    Bulk_quote* clone() const & {return new Bulk_quote(*this);}
    Bulk_quote* clone() &&
        {return new Bulk_quote(std::move(*this));}
    // other members as before
};

```

Because we have a copy and a move version of `add_item`, we defined lvalue and rvalue versions of `clone` (§ 13.6.3, p. 546). Each `clone` function allocates a new object of its own type. The `const` lvalue reference member copies itself into that newly allocated object; the rvalue reference member moves its own data.

Using `clone`, it is easy to write our new versions of `add_item`:

```

class Basket {
public:
    void add_item(const Quote& sale) // copy the given object
        { items.insert(std::shared_ptr<Quote>(sale.clone())); }
    void add_item(Quote&& sale) // move the given object
        { items.insert(
            std::shared_ptr<Quote>(std::move(sale).clone())); }
    // other members as before
};

```

Like `add_item` itself, `clone` is overloaded based on whether it is called on an lvalue or an rvalue. Thus, the first version of `add_item` calls the `const` lvalue version of `clone`, and the second version calls the rvalue reference version. Note that in the rvalue version, although the type of `sale` is an rvalue reference type, `sale` (like any other variable) is an lvalue (§ 13.6.1, p. 533). Therefore, we call `move` to bind an rvalue reference to `sale`.

Our `clone` function is also virtual. Whether the `Quote` or `Bulk_quote` function is run, depends (as usual) on the dynamic type of `sale`. Regardless of whether we copy or move the data, `clone` returns a pointer to a newly allocated object, of its own type. We bind a `shared_ptr` to that object and call `insert` to add this newly allocated object to `items`. Note that because `shared_ptr` supports the derived-to-base conversion (§ 15.2.2, p. 597), we can bind a `shared_ptr<Quote` to a `Bulk_quote*`.

### EXERCISES SECTION 15.8.1

**Exercise 15.30:** Write your own version of the `Basket` class and use it to compute prices for the same transactions as you used in the previous exercises.

## 15.9 Text Queries Revisited

As a final example of inheritance, we'll extend our text-query application from § 12.3 (p. 484). The classes we wrote in that section let us look for occurrences of a

given word in a file. We'd like to extend the system to support more complicated queries. In our examples, we'll run queries against the following simple story:

```
Alice Emma has long flowing red hair.  
Her Daddy says when the wind blows  
through her hair, it looks almost alive,  
like a fiery bird in flight.  
A beautiful fiery bird, he tells her,  
magical but untamed.  
"Daddy, shush, there is no such thing,"  
she tells him, at the same time wanting  
him to tell her more.  
Shyly, she asks, "I mean, Daddy, is there?"
```

Our system should support the following queries:

- Word queries find all the lines that match a given string:

```
Executing Query for: Daddy  
Daddy occurs 3 times  
(line 2) Her Daddy says when the wind blows  
(line 7) "Daddy, shush, there is no such thing,"  
(line 10) Shyly, she asks, "I mean, Daddy, is there?"
```

- Not queries, using the `~` operator, yield lines that don't match the query:

```
Executing Query for: ~(Alice)  
~(Alice) occurs 9 times  
(line 2) Her Daddy says when the wind blows  
(line 3) through her hair, it looks almost alive,  
(line 4) like a fiery bird in flight.  
 . . .
```

- Or queries, using the `|` operator, return lines matching either of two queries:

```
Executing Query for: (hair | Alice)  
(hair | Alice) occurs 2 times  
(line 1) Alice Emma has long flowing red hair.  
(line 3) through her hair, it looks almost alive,
```

- And queries, using the `&` operator, return lines matching both queries:

```
Executing query for: (hair & Alice)  
(hair & Alice) occurs 1 time  
(line 1) Alice Emma has long flowing red hair.
```

Moreover, we want to be able to combine these operations, as in

```
fiery & bird | wind
```

We'll use normal C++ precedence rules (§ 4.1.2, p. 136) to evaluate compound expressions such as this example. Thus, this query will match a line in which both `fiery` and `bird` appear or one in which `wind` appears:

```
Executing Query for: ((fiery & bird) | wind)
((fiery & bird) | wind) occurs 3 times
(line 2) Her Daddy says when the wind blows
(line 4) like a fiery bird in flight.
(line 5) A beautiful fiery bird, he tells her,
```

Our output will print the query, using parentheses to indicate the way in which the query was interpreted. As with our original implementation, our system will display lines in ascending order and will not display the same line more than once.

### 15.9.1 An Object-Oriented Solution

We might think that we should use the `TextQuery` class from § 12.3.2 (p. 487) to represent our word query and derive our other queries from that class.

However, this design would be flawed. To see why, consider a `Not` query. A `Word` query looks for a particular word. In order for a `Not` query to be a kind of `Word` query, we would have to be able to identify the word for which the `Not` query was searching. In general, there is no such word. Instead, a `Not` query has a query (a `Word` query or any other kind of query) whose value it negates. Similarly, an `And` query and an `Or` query have two queries whose results it combines.

This observation suggests that we model our different kinds of queries as independent classes that share a common base class:

```
WordQuery // Daddy
NotQuery // ~Alice
OrQuery // hair | Alice
AndQuery // hair & Alice
```

These classes will have only two operations:

- `eval`, which takes a `TextQuery` object and returns a `QueryResult`. The `eval` function will use the given `TextQuery` object to find the query's the matching lines.
- `rep`, which returns the string representation of the underlying query. This function will be used by `eval` to create a `QueryResult` representing the match and by the `operator<<` to print the query expressions.

### Abstract Base Class

As we've seen, our four query types are not related to one another by inheritance; they are conceptually siblings. Each class shares the same interface, which suggests that we'll need to define an abstract base class (§ 15.4, p. 610) to represent that interface. We'll name our abstract base class `Query_base`, indicating that its role is to serve as the root of our query hierarchy.

Our `Query_base` class will define `eval` and `rep` as pure virtual functions (§ 15.4, p. 610). Each of our classes that represents a particular kind of query must override these functions. We'll derive `WordQuery` and `NotQuery` directly from

**KEY CONCEPT: INHERITANCE VERSUS COMPOSITION**

The design of inheritance hierarchies is a complicated topic in its own right and well beyond the scope of this language Primer. However, there is one important design guide that is so fundamental that every programmer should be familiar with it.

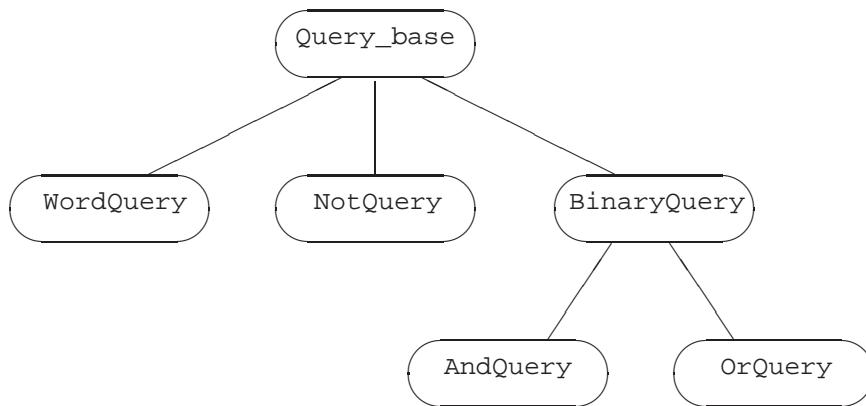
When we define a class as publicly inherited from another, the derived class should reflect an “Is A” relationship to the base class. In well-designed class hierarchies, objects of a publicly derived class can be used wherever an object of the base class is expected.

Another common relationship among types is a “Has A” relationship. Types related by a “Has A” relationship imply membership.

In our bookstore example, our base class represents the concept of a quote for a book sold at a stipulated price. Our `Bulk_quote` “is a” kind of quote, but one with a different pricing strategy. Our bookstore classes “have a” price and an ISBN.

`Query_base`. The `AndQuery` and `OrQuery` classes share one property that the other classes in our system do not: Each has two operands. To model this property, we’ll define another abstract base class, named `BinaryQuery`, to represent queries with two operands. The `AndQuery` and `OrQuery` classes will inherit from `BinaryQuery`, which in turn will inherit from `Query_base`. These decisions give us the class design represented in Figure 15.2.

Figure 15.2: `Query_base` Inheritance Hierarchy



### Hiding a Hierarchy in an Interface Class

Our program will deal with evaluating queries, not with building them. However, we need to be able to create queries in order to run our program. The simplest way to do so is to write C++ expressions to create the queries. For example, we’d like to generate the compound query previously described by writing code such as

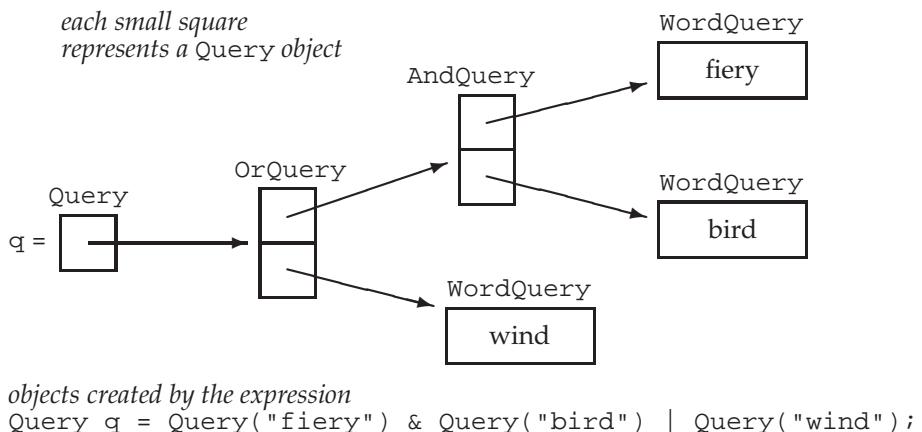
```
Query q = Query("fiery") & Query("bird") | Query("wind");
```

This problem description implicitly suggests that user-level code won't use the inherited classes directly. Instead, we'll define an interface class named `Query`, which will hide the hierarchy. The `Query` class will store a pointer to `Query_base`. That pointer will be bound to an object of a type derived from `Query_base`. The `Query` class will provide the same operations as the `Query_base` classes: `eval` to evaluate the associated query, and `rep` to generate a `string` version of the query. It will also define an overloaded output operator to display the associated query.

Users will create and manipulate `Query_base` objects only indirectly through operations on `Query` objects. We'll define three overloaded operators on `Query` objects, along with a `Query` constructor that takes a `string`. Each of these functions will dynamically allocate a new object of a type derived from `Query_base`:

- The `&` operator will generate a `Query` bound to a new `AndQuery`.
- The `|` operator will generate a `Query` bound to a new `OrQuery`.
- The `~` operator will generate a `Query` bound to a new `NotQuery`.
- The `Query` constructor that takes a `string` will generate a new `WordQuery`.

**Figure 15.3: Objects Created by `Query` Expressions**



## Understanding How These Classes Work

It is important to realize that much of the work in this application consists of building objects to represent the user's query. For example, an expression such as the one above generates the collection of interrelated objects illustrated in Figure 15.3.

Once the tree of objects is built up, evaluating (or generating the representation of) a query is basically a process (managed for us by the compiler) of following these links, asking each object to evaluate (or display) itself. For example, if we

call `eval` on `q` (i.e., on the root of the tree), that call asks the `OrQuery` to which `q` points to `eval` itself. Evaluating this `OrQuery` calls `eval` on its two operands—on the `AndQuery` and the `WordQuery` that looks for the word `wind`. Evaluating the `AndQuery` evaluates its two `WordQuery`s, generating the results for the words `fiery` and `bird`, respectively.

When new to object-oriented programming, it is often the case that the hardest part in understanding a program is understanding the design. Once you are thoroughly comfortable with the design, the implementation flows naturally. As an aid to understanding this design, we've summarized the classes used in this example in Table 15.1 (overleaf).

### EXERCISES SECTION 15.9.1

**Exercise 15.31:** Given that `s1`, `s2`, `s3`, and `s4` are all `strings`, determine what objects are created in the following expressions:

- (a) `Query(s1) | Query(s2) & ~ Query(s3);`
- (b) `Query(s1) | (Query(s2) & ~ Query(s3));`
- (c) `(Query(s1) & (Query(s2)) | (Query(s3) & Query(s4)));`

## 15.9.2 The `Query_base` and `Query` Classes

We'll start our implementation by defining the `Query_base` class:

```
// abstract class acts as a base class for concrete query types; all members are private
class Query_base {
    friend class Query;
protected:
    using line_no = TextQuery::line_no; // used in the eval functions
    virtual ~Query_base() = default;
private:
    // eval returns the QueryResult that matches this Query
    virtual QueryResult eval(const TextQuery&) const = 0;
    // rep is a string representation of the query
    virtual std::string rep() const = 0;
};
```

Both `eval` and `rep` are pure virtual functions, which makes `Query_base` an abstract base class (§ 15.4, p. 610). Because we don't intend users, or the derived classes, to use `Query_base` directly, `Query_base` has no public members. All use of `Query_base` will be through `Query` objects. We grant friendship to the `Query` class, because members of `Query` will call the virtuals in `Query_base`.

The protected member, `line_no`, will be used inside the `eval` functions. Similarly, the destructor is protected because it is used (implicitly) by the destructors in the derived classes.

| Table 15.1: Recap: Query Program Design        |                                                                                                                                                                                                                        |
|------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Query Program Interface Classes and Operations |                                                                                                                                                                                                                        |
| TextQuery                                      | Class that reads a given file and builds a lookup map. This class has a query operation that takes a string argument and returns a QueryResult representing the lines on which that string appears (§ 12.3.2, p. 487). |
| QueryResult                                    | Class that holds the results of a query operation (§ 12.3.2, p. 489).                                                                                                                                                  |
| Query                                          | Interface class that points to an object of a type derived from Query_base.                                                                                                                                            |
| Query q(s)                                     | Binds the Query q to a new WordQuery holding the string s.                                                                                                                                                             |
| q1 & q2                                        | Returns a Query bound to a new AndQuery object holding q1 and q2.                                                                                                                                                      |
| q1   q2                                        | Returns a Query bound to a new OrQuery object holding q1 and q2.                                                                                                                                                       |
| ~q                                             | Returns a Query bound to a new NotQuery object holding q.                                                                                                                                                              |
| Query Program Implementation Classes           |                                                                                                                                                                                                                        |
| Query_base                                     | Abstract base class for the query classes.                                                                                                                                                                             |
| WordQuery                                      | Class derived from Query_base that looks for a given word.                                                                                                                                                             |
| NotQuery                                       | Class derived from Query_base that represents the set of lines in which its Query operand does not appear.                                                                                                             |
| BinaryQuery                                    | Abstract base class derived from Query_base that represents queries with two Query operands.                                                                                                                           |
| OrQuery                                        | Class derived from BinaryQuery that returns the union of the line numbers in which its two operands appear.                                                                                                            |
| AndQuery                                       | Class derived from BinaryQuery that returns the intersection of the line numbers in which its two operands appear.                                                                                                     |

## The Query Class

The Query class provides the interface to (and hides) the Query\_base inheritance hierarchy. Each Query object will hold a shared\_ptr to a corresponding Query\_base object. Because Query is the only interface to the Query\_base classes, Query must define its own versions of eval and rep.

The Query constructor that takes a string will create a new WordQuery and bind its shared\_ptr member to that newly created object. The &, |, and ~ operators will create AndQuery, OrQuery, and NotQuery objects, respectively. These operators will return a Query object bound to its newly generated object. To support these operators, Query needs a constructor that takes a shared\_ptr to a Query\_base and stores its given pointer. We'll make this constructor private because we don't intend general user code to define Query\_base objects. Because this constructor is private, we'll need to make the operators friends.

Given the preceding design, the Query class itself is simple:

```
// interface class to manage the Query_base inheritance hierarchy
class Query {
    // these operators need access to the shared_ptr constructor
    friend Query operator~(const Query &);
    friend Query operator|(const Query&, const Query&);
    friend Query operator&(const Query&, const Query&);
```

```
public:  
    Query(const std::string&); // builds a new WordQuery  
    // interface functions: call the corresponding Query_base operations  
    QueryResult eval(const TextQuery &t) const  
        { return q->eval(t); }  
    std::string rep() const { return q->rep(); }  
private:  
    Query(std::shared_ptr<Query_base> query): q(query) { }  
    std::shared_ptr<Query_base> q;  
};
```

We start by naming as friends the operators that create `Query` objects. These operators need to be friends in order to use the `private` constructor.

In the `public` interface for `Query`, we declare, but cannot yet define, the constructor that takes a `string`. That constructor creates a `WordQuery` object, so we cannot define this constructor until we have defined the `WordQuery` class.

The other two `public` members represent the interface for `Query_base`. In each case, the `Query` operation uses its `Query_base` pointer to call the respective (virtual) `Query_base` operation. The actual version that is called is determined at run time and will depend on the type of the object to which `q` points.

## The Query Output Operator



The output operator is a good example of how our overall query system works:

```
std::ostream &  
operator<<(std::ostream &os, const Query &query)  
{  
    // Query::rep makes a virtual call through its Query_base pointer to rep()  
    return os << query.rep();  
}
```

When we print a `Query`, the output operator calls the (`public`) `rep` member of class `Query`. That function makes a virtual call through its pointer member to the `rep` member of the object to which this `Query` points. That is, when we write

```
Query andq = Query(sought1) & Query(sought2);  
cout << andq << endl;
```

the output operator calls `Query::rep` on `andq`. `Query::rep` in turn makes a virtual call through its `Query_base` pointer to the `Query_base` version of `rep`. Because `andq` points to an `AndQuery` object, that call will run `AndQuery::rep`.

### EXERCISES SECTION 15.9.2

**Exercise 15.32:** What happens when an object of type `Query` is copied, moved, assigned, and destroyed?

**Exercise 15.33:** What about objects of type `Query_base`?

### 15.9.3 The Derived Classes

The most interesting part of the classes derived from `Query_base` is how they are represented. The `WordQuery` class is most straightforward. Its job is to hold the search word.

The other classes operate on one or two operands. A `NotQuery` has a single operand, and `AndQuery` and `OrQuery` have two operands. In each of these classes, the operand(s) can be an object of any of the concrete classes derived from `Query_base`: A `NotQuery` can be applied to a `WordQuery`, an `AndQuery`, an `OrQuery`, or another `NotQuery`. To allow this flexibility, the operands must be stored as pointers to `Query_base`. That way we can bind the pointer to whichever concrete class we need.

However, rather than storing a `Query_base` pointer, our classes will themselves use a `Query` object. Just as user code is simplified by using the interface class, we can simplify our own class code by using the same class.

Now that we know the design for these classes, we can implement them.

#### The `WordQuery` Class

A `WordQuery` looks for a given string. It is the only operation that actually performs a query on the given `TextQuery` object:

```
class WordQuery: public Query_base {
    friend class Query; // Query uses the WordQuery constructor
    WordQuery(const std::string &s): query_word(s) { }
    // concrete class: WordQuery defines all inherited pure virtual functions
    QueryResult eval(const TextQuery &t) const
        { return t.query(query_word); }
    std::string rep() const { return query_word; }
    std::string query_word; // word for which to search
};
```

Like `Query_base`, `WordQuery` has no public members; `WordQuery` must make `Query` a friend in order to allow `Query` to access the `WordQuery` constructor.

Each of the concrete query classes must define the inherited pure virtual functions, `eval` and `rep`. We defined both operations inside the `WordQuery` class body: `eval` calls the `query` member of its given `TextQuery` parameter, which does the actual search in the file; `rep` returns the string that this `WordQuery` represents (i.e., `query_word`).

Having defined the `WordQuery` class, we can now define the `Query` constructor that takes a `string`:

```
inline
Query::Query(const std::string &s): q(new WordQuery(s)) { }
```

This constructor allocates a `WordQuery` and initializes its pointer member to point to that newly allocated object.

#### The `NotQuery` Class and the `~` Operator

The `~` operator generates a `NotQuery`, which holds a `Query`, which it negates:

```

class NotQuery: public Query_base {
    friend Query operator~(const Query &);
    NotQuery(const Query &q): query(q) { }
    // concrete class: NotQuery defines all inherited pure virtual functions
    std::string rep() const {return "~(" + query.rep() + ")"; }
    QueryResult eval(const TextQuery&) const;
    Query query;
};

inline Query operator~(const Query &operand)
{
    return std::shared_ptr<Query_base>(new NotQuery(operand));
}

```

Because the members of `NotQuery` are all `private`, we start by making the `~` operator a friend. To `rep` a `NotQuery`, we concatenate the `~` symbol to the representation of the underlying `Query`. We parenthesize the output to ensure that precedence is clear to the reader.

It is worth noting that the call to `rep` in `NotQuery`'s own `rep` member ultimately makes a virtual call to `rep`: `query.rep()` is a nonvirtual call to the `rep` member of the `Query` class. `Query::rep` in turn calls `q->rep()`, which is a virtual call through its `Query_base` pointer.

The `~` operator dynamically allocates a new `NotQuery` object. The return (implicitly) uses the `Query` constructor that takes a `shared_ptr<Query_base>`. That is, the `return` statement is equivalent to

```

// allocate a new NotQuery object
// bind the resulting NotQuery pointer to a shared_ptr<Query_base>
shared_ptr<Query_base> tmp(new NotQuery(expr));
return Query(tmp); // use the Query constructor that takes a shared_ptr

```

The `eval` member is complicated enough that we will implement it outside the class body. We'll define the `eval` functions in § 15.9.4 (p. 647).

## The BinaryQuery Class

The `BinaryQuery` class is an abstract base class that holds the data needed by the query types that operate on two operands:

```

class BinaryQuery: public Query_base {
protected:
    BinaryQuery(const Query &l, const Query &r, std::string s):
        lhs(l), rhs(r), opSym(s) { }
    // abstract class: BinaryQuery doesn't define eval
    std::string rep() const { return "(" + lhs.rep() + " "
                                + opSym + " "
                                + rhs.rep() + ")"; }
    Query lhs, rhs;      // right- and left-hand operands
    std::string opSym;   // name of the operator
};

```

The data in a `BinaryQuery` are the two `Query` operands and the corresponding operator symbol. The constructor takes the two operands and the operator symbol, each of which it stores in the corresponding data members.

To `rep` a `BinaryOperator`, we generate the parenthesized expression consisting of the representation of the left-hand operand, followed by the operator, followed by the representation of the right-hand operand. As when we displayed a `NotQuery`, the calls to `rep` ultimately make virtual calls to the `rep` function of the `Query_base` objects to which `lhs` and `rhs` point.



The `BinaryQuery` class does not define the `eval` function and so inherits a pure virtual. Thus, `BinaryQuery` is also an abstract base class, and we cannot create objects of `BinaryQuery` type.

## The `AndQuery` and `OrQuery` Classes and Associated Operators

The `AndQuery` and `OrQuery` classes, and their corresponding operators, are quite similar to one another:

```
class AndQuery: public BinaryQuery {
    friend Query operator&(const Query&, const Query&);
    AndQuery(const Query &left, const Query &right):
        BinaryQuery(left, right, "&") { }
    // concrete class: AndQuery inherits rep and defines the remaining pure virtual
    QueryResult eval(const TextQuery&) const;
};

inline Query operator&(const Query &lhs, const Query &rhs)
{
    return std::shared_ptr<Query_base>(new AndQuery(lhs, rhs));
}

class OrQuery: public BinaryQuery {
    friend Query operator|(const Query&, const Query&);
    OrQuery(const Query &left, const Query &right):
        BinaryQuery(left, right, "|") { }
    QueryResult eval(const TextQuery&) const;
};

inline Query operator|(const Query &lhs, const Query &rhs)
{
    return std::shared_ptr<Query_base>(new OrQuery(lhs, rhs));
}
```

These classes make the respective operator a friend and define a constructor to create their `BinaryQuery` base part with the appropriate operator. They inherit the `BinaryQuery` definition of `rep`, but each overrides the `eval` function.

Like the `~` operator, the `&` and `|` operators return a `shared_ptr` bound to a newly allocated object of the corresponding type. That `shared_ptr` gets converted to `Query` as part of the return statement in each of these operators.

**EXERCISES SECTION 15.9.3**

**Exercise 15.34:** For the expression built in Figure 15.3 (p. 638):

- List the constructors executed in processing that expression.
- List the calls to `rep` that are made from `cout << q`.
- List the calls to `eval` made from `q.eval()`.

**Exercise 15.35:** Implement the `Query` and `Query_base` classes, including a definition of `rep` but omitting the definition of `eval`.

**Exercise 15.36:** Put print statements in the constructors and `rep` members and run your code to check your answers to (a) and (b) from the first exercise.

**Exercise 15.37:** What changes would your classes need if the derived classes had members of type `shared_ptr<Query_base>` rather than of type `Query`?

**Exercise 15.38:** Are the following declarations legal? If not, why not? If so, explain what the declarations mean.

```
BinaryQuery a = Query("fiery") & Query("bird");
AndQuery b = Query("fiery") & Query("bird");
OrQuery c = Query("fiery") & Query("bird");
```

## 15.9.4 The `eval` Functions

The `eval` functions are the heart of our query system. Each of these functions calls `eval` on its operand(s) and then applies its own logic: The `OrQuery eval` operation returns the union of the results of its two operands; `AndQuery` returns the intersection. The `NotQuery` is more complicated: It must return the line numbers that are not in its operand's set.

To support the processing in the `eval` functions, we need to use the version of `QueryResult` that defines the members we added in the exercises to § 12.3.2 (p. 490). We'll assume that `QueryResult` has `begin` and `end` members that will let us iterate through the set of line numbers that the `QueryResult` holds. We'll also assume that `QueryResult` has a member named `get_file` that returns a `shared_ptr` to the underlying file on which the query was executed.



Our `Query` classes use members defined for `QueryResult` in the exercises to § 12.3.2 (p. 490).

### `OrQuery::eval`

An `OrQuery` represents the union of the results for its two operands, which we obtain by calling `eval` on each of its operands. Because these operands are `Query` objects, calling `eval` is a call to `Query::eval`, which in turn makes a virtual call to `eval` on the underlying `Query_base` object. Each of these calls yields a `QueryResult` representing the line numbers in which its operand appears. We'll combine those line numbers into a new set:

```

// returns the union of its operands' result sets
QueryResult
OrQuery::eval(const TextQuery& text) const
{
    // virtual calls through the Query members, lhs and rhs
    // the calls to eval return the QueryResult for each operand
    auto right = rhs.eval(text), left = lhs.eval(text);
    // copy the line numbers from the left-hand operand into the result set
    auto ret_lines =
        make_shared<set<line_no>>(left.begin(), left.end());
    // insert lines from the right-hand operand
    ret_lines->insert(right.begin(), right.end());
    // return the new QueryResult representing the union of lhs and rhs
    return QueryResult(rep(), ret_lines, left.get_file());
}

```

We initialize `ret_lines` using the set constructor that takes a pair of iterators. The `begin` and `end` members of a `QueryResult` return iterators into that object's set of line numbers. So, `ret_lines` is created by copying the elements from `left`'s set. We next call `insert` on `ret_lines` to insert the elements from `right`. After this call, `ret_lines` contains the line numbers that appear in either `left` or `right`.

The `eval` function ends by building and returning a `QueryResult` representing the combined match. The `QueryResult` constructor (§ 12.3.2, p. 489) takes three arguments: a string representing the query, a `shared_ptr` to the set of matching line numbers, and a `shared_ptr` to the vector that represents the input file. We call `rep` to generate the string and `get_file` to obtain the `shared_ptr` to the file. Because both `left` and `right` refer to the same file, it doesn't matter which of these we use for `get_file`.

### **AndQuery::eval**

The `AndQuery` version of `eval` is similar to the `OrQuery` version, except that it calls a library algorithm to find the lines in common to both queries:

```

// returns the intersection of its operands' result sets
QueryResult
AndQuery::eval(const TextQuery& text) const
{
    // virtual calls through the Query operands to get result sets for the operands
    auto left = lhs.eval(text), right = rhs.eval(text);
    // set to hold the intersection of left and right
    auto ret_lines = make_shared<set<line_no>>();
    // writes the intersection of two ranges to a destination iterator
    // destination iterator in this call adds elements to ret
    set_intersection(left.begin(), left.end(),
                    right.begin(), right.end(),
                    inserter(*ret_lines, ret_lines->begin()));
    return QueryResult(rep(), ret_lines, left.get_file());
}

```

Here we use the library `set_intersection` algorithm, which is described in Appendix A.2.8 (p. 880), to merge these two sets.

The `set_intersection` algorithm takes five iterators. It uses the first four to denote two input sequences (§ 10.5.2, p. 413). Its last argument denotes a destination. The algorithm writes the elements that appear in both input sequences into the destination.

In this call we pass an `insert` iterator (§ 10.4.1, p. 401) as the destination. When `set_intersection` writes to this iterator, the effect will be to insert a new element into `ret_lines`.

Like the `OrQuery eval` function, this one ends by building and returning a `QueryResult` representing the combined match.

### **NotQuery::eval**

`NotQuery` finds each line of the text within which the operand is not found:

```
// returns the lines not in its operand's result set
QueryResult
NotQuery::eval(const TextQuery& text) const
{
    // virtual call to eval through the Query operand
    auto result = query.eval(text);

    // start out with an empty result set
    auto ret_lines = make_shared<set<line_no>>();

    // we have to iterate through the lines on which our operand appears
    auto beg = result.begin(), end = result.end();

    // for each line in the input file, if that line is not in result,
    // add that line number to ret_lines
    auto sz = result.get_file()->size();
    for (size_t n = 0; n != sz; ++n) {
        // if we haven't processed all the lines in result
        // check whether this line is present
        if (beg == end || *beg != n)
            ret_lines->insert(n); // if not in result, add this line
        else if (beg != end)
            ++beg; // otherwise get the next line number in result if there is one
    }
    return QueryResult(rep(), ret_lines, result.get_file());
}
```

As in the other `eval` functions, we start by calling `eval` on this object's operand. That call returns the `QueryResult` containing the line numbers on which the operand appears, but we want the line numbers on which the operand does not appear. That is, we want every line in the file that is not already in `result`.

We generate that `set` by iterating through sequential integers up to the size of the input file. We'll put each number that is not in `result` into `ret_lines`. We position `beg` and `end` to denote the first and one past the last elements in `result`. That object is a `set`, so when we iterate through it, we'll obtain the line numbers in ascending order.

The loop body checks whether the current number is in `result`. If not, we add that number to `ret_lines`. If the number is in `result`, we increment `beg`, which is our iterator into `result`.

Once we've processed all the line numbers, we return a `QueryResult` containing `ret_lines`, along with the results of running `rep` and `get_file` as in the previous `eval` functions.

### EXERCISES SECTION 15.9.4

**Exercise 15.39:** Implement the `Query` and `Query_base` classes. Test your application by evaluating and printing a query such as the one in Figure 15.3 (p. 638).

**Exercise 15.40:** In the `OrQuery eval` function what would happen if its `rhs` member returned an empty set? What if its `lhs` member did so? What if both `rhs` and `lhs` returned empty sets?

**Exercise 15.41:** Reimplement your classes to use built-in pointers to `Query_base` rather than `shared_ptrs`. Remember that your classes will no longer be able to use the synthesized copy-control members.

**Exercise 15.42:** Design and implement one of the following enhancements:

- (a) Print words only once per sentence rather than once per line.
- (b) Introduce a history system in which the user can refer to a previous query by number, possibly adding to it or combining it with another.
- (c) Allow the user to limit the results so that only matches in a given range of lines are displayed.

*The library* constitutes nearly two-thirds of the text of the new standard. Although we cannot cover every library facility in depth, there remain a few library facilities that are likely to be of use in many applications: tuples, bitsets, regular expressions, and random numbers. We'll also look at some additional IO library capabilities: format control, unformatted IO, and random access.

## 17.1 The tuple Type

A **tuple** is a template that is similar to a pair (§ 11.2.3, p. 426). Each pair type has different types for its members, but every pair always has exactly two members.

C++  
11

A tuple also has members whose types vary from one tuple type to another, but a tuple can have any number of members. Each distinct tuple type has a fixed number of members, but the number of members in one tuple type can differ from the number of members in another.

A tuple is most useful when we want to combine some data into a single object but do not want to bother to define a data structure to represent those data. Table 17.1 lists the operations that tuples support. The tuple type, along with its companion types and functions, are defined in the tuple header.

*Note*

A tuple can be thought of as a “quick and dirty” data structure.

### 17.1.1 Defining and Initializing tuples

When we define a tuple, we name the type(s) of each of its members:

```
tuple<size_t, size_t, size_t> threeD; // all three members set to 0
tuple<string, vector<double>, int, list<int>>
    someVal("constants", {3.14, 2.718}, 42, {0,1,2,3,4,5});
```

When we create a tuple object, we can use the default tuple constructor, which value initializes (§ 3.3.1, p. 98) each member, or we can supply an initializer for each member as we do in the initialization of someVal. This tuple constructor is explicit (§ 7.5.4, p. 296), so we must use the direct initialization syntax:

```
tuple<size_t, size_t, size_t> threeD = {1,2,3}; // error
tuple<size_t, size_t, size_t> threeD{1,2,3}; // ok
```

Alternatively, similar to the make\_pair function (§ 11.2.3, p. 428), the library defines a make\_tuple function that generates a tuple object:

```
// tuple that represents a bookstore transaction: ISBN, count, price per book
auto item = make_tuple("0-999-78345-X", 3, 20.00);
```

Like make\_pair, the make\_tuple function uses the types of the supplied initializers to infer the type of the tuple. In this case, item is a tuple whose type is tuple<const char\*, int, double>.

**Table 17.1: Operations on tuples**

|                                                               |                                                                                                                                                                                                                         |
|---------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>tuple&lt;T1, T2, ..., Tn&gt; t;</code>                  | t is a tuple with as many members as there are types T1 ... Tn. The members are value initialized (§ 3.3.1, p. 98).                                                                                                     |
| <code>tuple&lt;T1, T2, ..., Tn&gt; t(v1, v2, ..., vn);</code> | t is a tuple with types T1 ... Tn in which each member is initialized from the corresponding initializer, vi. This constructor is explicit (§ 7.5.4, p. 296).                                                           |
| <code>make_tuple(v1, v2, ..., vn)</code>                      | Returns a tuple initialized from the given initializers. The type of the tuple is inferred from the types of the initializers.                                                                                          |
| <code>t1 == t2</code>                                         | Two tuples are equal if they have the same number of members and if each pair of members are equal. Uses each member's underlying == operator. Once a member is found to be unequal, subsequent members are not tested. |
| <code>t1 relop t2</code>                                      | Relational operations on tuples using dictionary ordering (§ 9.2.7, p. 340). The tuples must have the same number of members. Members of t1 are compared with the corresponding members from t2 using the < operator    |
| <code>get&lt;i&gt;(t)</code>                                  | Returns a reference to the i <sup>th</sup> data member of t; if t is an lvalue, the result is an lvalue reference; otherwise, it is an rvalue reference. All members of a tuple are public.                             |
| <code>tuple_size&lt;tupleType&gt;::value</code>               | A class template that can be instantiated by a tuple type and has a public constexpr static data member named value of type size_t that is number of members in the specified tuple type.                               |
| <code>tuple_element&lt;i, tupleType&gt;::type</code>          | A class template that can be instantiated by an integral constant and a tuple type and has a public member named type that is the type of the specified members in the specified tuple type.                            |

## Accessing the Members of a tuple

A pair always has two members, which makes it possible for the library to give these members names (i.e., `first` and `second`). No such naming convention is possible for `tuple` because there is no limit on the number of members a tuple type can have. As a result, the members are unnamed. Instead, we access the members of a tuple through a library function template named `get`. To use `get` we must specify an explicit template argument (§ 16.2.2, p. 682), which is the position of the member we want to access. We pass a `tuple` object to `get`, which returns a reference to the specified member:

```
auto book = get<0>(item);           // returns the first member of item
auto cnt = get<1>(item);            // returns the second member of item
auto price = get<2>(item)/cnt;      // returns the last member of item
get<2>(item) *= 0.8;                // apply 20% discount
```

The value inside the brackets must be an integral constant expression (§ 2.4.4, p. 65). As usual, we count from 0, meaning that `get<0>` is the first member.

If we have a `tuple` whose precise type details we don't know, we can use two auxilliary class templates to find the number and types of the `tuple`'s members:

```

typedef decltype(item) trans; // trans is the type of item
// returns the number of members in object's of type trans
size_t sz = tuple_size<trans>::value; // returns 3
// cnt has the same type as the second member in item
tuple_element<1, trans>::type cnt = get<1>(item); // cnt is an int

```

To use `tuple_size` or `tuple_element`, we need to know the type of a tuple object. As usual, the easiest way to determine an object's type is to use `decltype` (§ 2.5.3, p. 70). Here, we use `decltype` to define a type alias for the type of `item`, which we use to instantiate both templates.

`tuple_size` has a public static data member named `value` that is the number of members in the specified tuple. The `tuple_element` template takes an index as well as a tuple type. `tuple_element` has a public type member named `type` that is the type of the specified member of the specified tuple type. Like `get`, `tuple_element` uses indices starting at 0.

## Relational and Equality Operators

The `tuple` relational and equality operators behave similarly to the corresponding operations on containers (§ 9.2.7, p. 340). These operators execute pairwise on the members of the left-hand and right-hand tuples. We can compare two tuples only if they have the same number of members. Moreover, to use the equality or inequality operators, it must be legal to compare each pair of members using the `==` operator; to use the relational operators, it must be legal to use `<`. For example:

```

tuple<string, string> duo("1", "2");
tuple<size_t, size_t> twoD(1, 2);
bool b = (duo == twoD); // error: can't compare a size_t and a string
tuple<size_t, size_t, size_t> threeD(1, 2, 3);
b = (twoD < threeD); // error: differing number of members
tuple<size_t, size_t> origin(0, 0);
b = (origin < twoD); // ok: b is true

```



Because `tuple` defines the `<` and `==` operators, we can pass sequences of tuples to the algorithms and can use a `tuple` as key type in an ordered container.

### EXERCISES SECTION 17.1.1

**Exercise 17.1:** Define a `tuple` that holds three `int` values and initialize the members to 10, 20, and 30.

**Exercise 17.2:** Define a `tuple` that holds a `string`, a `vector<string>`, and a `pair<string, int>`.

**Exercise 17.3:** Rewrite the `TextQuery` programs from § 12.3 (p. 484) to use a `tuple` instead of the `QueryResult` class. Explain which design you think is better and why.

### 17.1.2 Using a `tuple` to Return Multiple Values

A common use of `tuple` is to return multiple values from a function. For example, our bookstore might be one of several stores in a chain. Each store would have a transaction file that holds data on each book that the store recently sold. We might want to look at the sales for a given book in all the stores.

We'll assume that we have a file of transactions for each store. Each of these per-store transaction files will contain all the transactions for each book grouped together. We'll further assume that some other function reads these transaction files, builds a `vector<Sales_data>` for each store, and puts those vectors in a `vector` of vectors:

```
// each element in files holds the transactions for a particular store
vector<vector<Sales_data>> files;
```

We'll write a function that will search `files` looking for the stores that sold a given book. For each store that has a matching transaction, we'll create a `tuple` to hold the index of that store and two iterators. The index will be the position of the matching store in `files`. The iterators will mark the first and one past the last record for the given book in that store's `vector<Sales_data>`.

### A Function That Returns a `tuple`

We'll start by writing the function to find a given book. This function's arguments are the `vector` of `vectors` just described, and a `string` that represents the book's ISBN. Our function will return a `vector` of `tuples` that will have an entry for each store with at least one sale for the given book:

```
// matches has three members: an index of a store and iterators into that store's vector
typedef tuple<vector<Sales_data>::size_type,
              vector<Sales_data>::const_iterator,
              vector<Sales_data>::const_iterator> matches;

// files holds the transactions for every store
// findBook returns a vector with an entry for each store that sold the given book
vector<matches>
findBook(const vector<vector<Sales_data>> &files,
         const string &book)
{
    vector<matches> ret; // initially empty
    // for each store find the range of matching books, if any
    for (auto it = files.cbegin(); it != files.cend(); ++it) {
        // find the range of Sales_data that have the same ISBN
        auto found = equal_range(it->cbegin(), it->cend(),
                                book, compareIsbn);
        if (found.first != found.second) // this store had sales
            // remember the index of this store and the matching range
            ret.push_back(make_tuple(it - files.cbegin(),
                                    found.first, found.second));
    }
    return ret; // empty if no matches found
}
```

The `for` loop iterates through the elements in `files`. Those elements are themselves vectors. Inside the `for` we call a library algorithm named `equal_range`, which operates like the associative container member of the same name (§ 11.3.5, p. 439). The first two arguments to `equal_range` are iterators denoting an input sequence (§ 10.1, p. 376). The third argument is a value. By default, `equal_range` uses the `<` operator to compare elements. Because `Sales_data` does not have a `<` operator, we pass a pointer to the `compareIsbn` function (§ 11.2.2, p. 425).

The `equal_range` algorithm returns a pair of iterators that denote a range of elements. If book is not found, then the iterators will be equal, indicating that the range is empty. Otherwise, the first member of the returned pair will denote the first matching transaction and second will be one past the last.

## Using a tuple Returned by a Function

Once we have built our vector of stores with matching transactions, we need to process these transactions. In this program, we'll report the total sales results for each store that has a matching sale:

```
void reportResults(istream &in, ostream &os,
                   const vector<vector<Sales_data>> &files)
{
    string s;      // book to look for
    while (in >> s) {
        auto trans = findBook(files, s); // stores that sold this book
        if (trans.empty()) {
            cout << s << " not found in any stores" << endl;
            continue; // get the next book to look for
        }
        for (const auto &store : trans) // for every store with a sale
            // get<n> returns the specified member from the tuple in store
            os << "store " << get<0>(store) << " sales: "
            << accumulate(get<1>(store), get<2>(store),
                          Sales_data(s))
            << endl;
    }
}
```

The `while` loop repeatedly reads the `istream` named `in` to get the next book to process. We call `findBook` to see if `s` is present, and assign the results to `trans`. We use `auto` to simplify writing the type of `trans`, which is a vector of tuples.

If `trans` is empty, there were no sales for `s`. In this case, we print a message and return to the `while` to get the next book to look for.

The `for` loop binds `store` to each element in `trans`. Because we don't intend to change the elements in `trans`, we declare `store` as a reference to `const`. We use `get` to print the relevant data: `get<0>` is the index of the corresponding store, `get<1>` is the iterator denoting the first transaction, and `get<2>` is the iterator one past the last.

Because `Sales_data` defines the addition operator (§ 14.3, p. 560), we can use the library `accumulate` algorithm (§ 10.2.1, p. 379) to sum the transactions. We

pass a `Sales_data` object initialized by the `Sales_data` constructor that takes a string (§ 7.1.4, p. 264) as the starting point for the summation. That constructor initializes the `bookNo` member from the given string and the `units_sold` and `revenue` members to zero.

### EXERCISES SECTION 17.1.2

**Exercise 17.4:** Write and test your own version of the `findBook` function.

**Exercise 17.5:** Rewrite `findBook` to return a pair that holds an index and a pair of iterators.

**Exercise 17.6:** Rewrite `findBook` so that it does not use `tuple` or `pair`.

**Exercise 17.7:** Explain which version of `findBook` you prefer and why.

**Exercise 17.8:** What would happen if we passed `Sales_data()` as the third parameter to `accumulate` in the last code example in this section?

## 17.2 The `bitset` Type

In § 4.8 (p. 152) we covered the built-in operators that treat an integral operand as a collection of bits. The standard library defines the `bitset` class to make it easier to use bit operations and possible to deal with collections of bits that are larger than the longest integral type. The `bitset` class is defined in the `bitset` header.

### 17.2.1 Defining and Initializing `bitsets`

Table 17.2 (overleaf) lists the constructors for `bitset`. The `bitset` class is a class template that, like the `array` class, has a fixed size (§ 9.2.4, p. 336). When we define a `bitset`, we say how many bits the `bitset` will contain:

```
bitset<32> bitvec(1U); // 32 bits; low-order bit is 1, remaining bits are 0
```

The size must be a constant expression (§ 2.4.4, p. 65). This statement defines `bitvec` as a `bitset` that holds 32 bits. Just as with the elements of a `vector`, the bits in a `bitset` are not named. Instead, we refer to them positionally. The bits are numbered starting at 0. Thus, `bitvec` has bits numbered 0 through 31. The bits starting at 0 are referred to as the **low-order** bits, and those ending at 31 are referred to as **high-order** bits.

#### Initializing a `bitset` from an `unsigned` Value

When we use an integral value as an initializer for a `bitset`, that value is converted to `unsigned long long` and is treated as a bit pattern. The bits in the `bitset` are a copy of that pattern. If the size of the `bitset` is greater than the number of bits in an `unsigned long long`, then the remaining high-order bits

| Table 17.2: Ways to Initialize a <code>bitset</code>                                                                                                                                                             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>bitset&lt;n&gt; b;</code>                                                                                                                                                                                  | <code>b</code> has <code>n</code> bits; each bit is 0. This constructor is a <code>constexpr</code> (§ 7.5.6, p. 299).                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| <code>bitset&lt;n&gt; b(u);</code>                                                                                                                                                                               | <code>b</code> is a copy of the <code>n</code> low-order bits of <code>unsigned long long</code> value <code>u</code> . If <code>n</code> is greater than the size of an <code>unsigned long long</code> , the high-order bits beyond those in the <code>unsigned long long</code> are set to zero. This constructor is a <code>constexpr</code> (§ 7.5.6, p. 299).                                                                                                                                                                                                                                                              |
| <code>bitset&lt;n&gt; b(s, pos, m, zero, one);</code>                                                                                                                                                            | <code>b</code> is a copy of the <code>m</code> characters from the <code>string</code> <code>s</code> starting at position <code>pos</code> . <code>s</code> may contain only the characters <code>zero</code> and <code>one</code> ; if <code>s</code> contains any other character, throws <code>invalid_argument</code> . The characters are stored in <code>b</code> as <code>zero</code> and <code>one</code> , respectively. <code>pos</code> defaults to 0, <code>m</code> defaults to <code>string::npos</code> , <code>zero</code> defaults to ' <code>0</code> ', and <code>one</code> defaults to ' <code>1</code> '. |
| <code>bitset&lt;n&gt; b(cp, pos, m, zero, one);</code>                                                                                                                                                           | Same as the previous constructor, but copies from the character array to which <code>cp</code> points. If <code>m</code> is not supplied, then <code>cp</code> must point to a C-style string. If <code>m</code> is supplied, there must be at least <code>m</code> characters that are <code>zero</code> or <code>one</code> starting at <code>cp</code> .                                                                                                                                                                                                                                                                      |
| <b>The constructors that take a <code>string</code> or character pointer are <code>explicit</code> (§ 7.5.4, p. 296). The ability to specify alternate characters for 0 and 1 was added in the new standard.</b> |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |

are set to zero. If the size of the `bitset` is less than that number of bits, then only the low-order bits from the given value are used; the high-order bits beyond the size of the `bitset` object are discarded:

```
// bitvec1 is smaller than the initializer; high-order bits from the initializer are discarded
bitset<13> bitvec1(0xbeef); // bits are 1111011101111
// bitvec2 is larger than the initializer; high-order bits in bitvec2 are set to zero
bitset<20> bitvec2(0xbeef); // bits are 0000101111011101111
// on machines with 64-bit long long OULL is 64 bits of 0, so ~OULL is 64 ones
bitset<128> bitvec3(~OULL); // bits 0...63 are one; 63...127 are zero
```

### Initializing a `bitset` from a `string`

We can initialize a `bitset` from either a `string` or a pointer to an element in a character array. In either case, the characters represent the bit pattern directly. As usual, when we use strings to represent numbers, the characters with the lowest indices in the string correspond to the high-order bits, and vice versa:

```
bitset<32> bitvec4("1100"); // bits 2 and 3 are 1, all others are 0
```

If the `string` contains fewer characters than the size of the `bitset`, the high-order bits are set to zero.

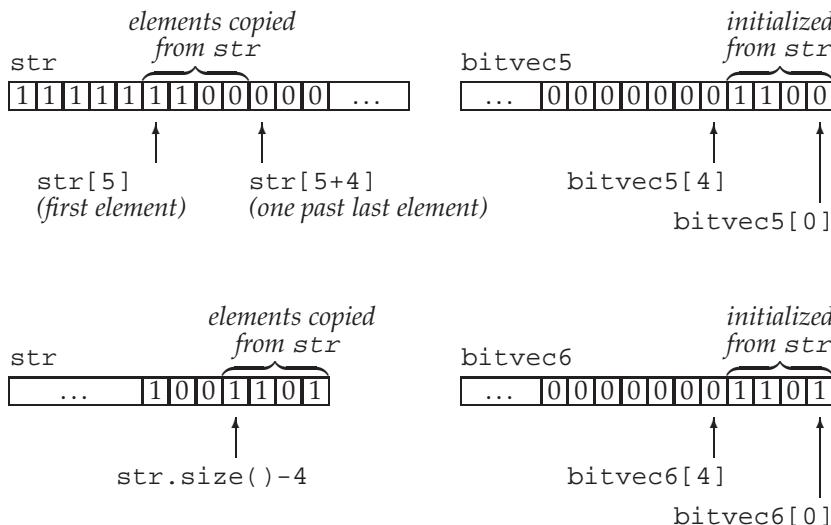


The indexing conventions of `strings` and `bitsets` are inversely related: The character in the `string` with the highest subscript (the right-most character) is used to initialize the low-order bit in the `bitset` (the bit with subscript 0). When you initialize a `bitset` from a `string`, it is essential to remember this difference.

We need not use the entire string as the initial value for the `bitset`. Instead, we can use a substring as the initializer:

```
string str("1111111000000011001101");
bitset<32> bitvec5(str, 5, 4); // four bits starting at str[5], 1100
bitset<32> bitvec6(str, str.size()-4); // use last four characters
```

Here `bitvec5` is initialized by the substring in `str` starting at `str[5]` and continuing for four positions. As usual, the right-most character of the substring represents the lowest-order bit. Thus, `bitvec5` is initialized with bit positions 3 through 0 set to 1100 and the remaining bits set to 0. The initializer for `bitvec6` passes a string and a starting point, so `bitvec6` is initialized from the characters in `str` starting four from the end of `str`. The remainder of the bits in `bitvec6` are initialized to zero. We can view these initializations as



### EXERCISES SECTION 17.2.1

**Exercise 17.9:** Explain the bit pattern each of the following `bitset` objects contains:

- `bitset<64> bitvec(32);`
- `bitset<32> bv(1010101);`
- `string bstr; cin >> bstr; bitset<8>bv(bstr);`

### 17.2.2 Operations on `bitsets`

The `bitset` operations (Table 17.3 (overleaf)) define various ways to test or set one or more bits. The `bitset` class also supports the bitwise operators that we covered in § 4.8 (p. 152). The operators have the same meaning when applied to `bitset` objects as the built-in operators have when applied to `unsigned` operands.

**Table 17.3: `bitset` Operations**

|                                     |                                                                                                                                                                                                                                                 |
|-------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>b.any()</code>                | Is any bit in <code>b</code> on?                                                                                                                                                                                                                |
| <code>b.all()</code>                | Are all the bits in <code>b</code> on?                                                                                                                                                                                                          |
| <code>b.none()</code>               | Are no bits in <code>b</code> on?                                                                                                                                                                                                               |
| <code>b.count()</code>              | Number of bits in <code>b</code> that are on.                                                                                                                                                                                                   |
| <code>b.size()</code>               | A <code>constexpr</code> function (§ 2.4.4, p. 65) that returns the number of bits in <code>b</code> .                                                                                                                                          |
| <code>b.test(pos)</code>            | Returns <code>true</code> if bit at position <code>pos</code> is on, <code>false</code> otherwise.                                                                                                                                              |
| <code>b.set(pos, v)</code>          | Sets the bit at position <code>pos</code> to the <code>bool</code> value <code>v</code> . <code>v</code> defaults to <code>true</code> . If no arguments, turns on all the bits in <code>b</code> .                                             |
| <code>b.set()</code>                | Turns off the bit at position <code>pos</code> or turns off all the bits in <code>b</code> .                                                                                                                                                    |
| <code>b.reset(pos)</code>           | Changes the state of the bit at position <code>pos</code> or of every bit in <code>b</code> .                                                                                                                                                   |
| <code>b.reset()</code>              |                                                                                                                                                                                                                                                 |
| <code>b.flip(pos)</code>            |                                                                                                                                                                                                                                                 |
| <code>b.flip()</code>               |                                                                                                                                                                                                                                                 |
| <code>b[pos]</code>                 | Gives access to the bit in <code>b</code> at position <code>pos</code> ; if <code>b</code> is <code>const</code> , then <code>b[pos]</code> returns a <code>bool</code> value <code>true</code> if the bit is on, <code>false</code> otherwise. |
| <code>b.to_ulong()</code>           | Returns an <code>unsigned long</code> or an <code>unsigned long long</code> with the same bits as in <code>b</code> . Throws <code>overflow_error</code> if the bit pattern in <code>b</code> won't fit in the indicated result type.           |
| <code>b.to_string(zero, one)</code> | Returns a <code>string</code> representing the bit pattern in <code>b</code> . <code>zero</code> and <code>one</code> default to ' <code>'0'</code> ' and ' <code>'1'</code> ' and are used to represent the bits 0 and 1 in <code>b</code> .   |
| <code>os &lt;&lt; b</code>          | Prints the bits in <code>b</code> as the characters 1 or 0 to the stream <code>os</code> .                                                                                                                                                      |
| <code>is &gt;&gt; b</code>          | Reads characters from <code>is</code> into <code>b</code> . Reading stops when the next character is not a 1 or 0 or when <code>b.size()</code> bits have been read.                                                                            |

Several operations—`count`, `size`, `all`, `any`, and `none`—take no arguments and return information about the state of the entire `bitset`. Others—`set`, `reset`, and `flip`—change the state of the `bitset`. The members that change the `bitset` are overloaded. In each case, the version that takes no arguments applies the given operation to the entire set; the versions that take a position apply the operation to the given bit:

```
bitset<32> bitvec(1U); // 32 bits; low-order bit is 1, remaining bits are 0
bool is_set = bitvec.any();           // true, one bit is set
bool is_not_set = bitvec.none();     // false, one bit is set
bool all_set = bitvec.all();         // false, only one bit is set
size_t onBits = bitvec.count();      // returns 1
size_t sz = bitvec.size();          // returns 32
bitvec.flip();                     // reverses the value of all the bits in bitvec
bitvec.reset();                    // sets all the bits to 0
bitvec.set();                      // sets all the bits to 1
```

The `any` operation returns `true` if one or more bits of the `bitset` object are turned on—that is, are equal to 1. Conversely, `none` returns `true` if all the bits are zero. The new standard introduced the `all` operation, which returns `true` if all the bits are on. The `count` and `size` operations return a `size_t` (§ 3.5.2, p. 116) equal to

the number of bits that are set, or the total number of bits in the object, respectively. The `size` function is a `constexpr` and so can be used where a constant expression is required (§ 2.4.4, p. 65).

The `flip`, `set`, `reset`, and `test` members let us read or write the bit at a given position:

```
bitvec.flip(0);    // reverses the value of the first bit
bitvec.set(bitvec.size() - 1); // turns on the last bit
bitvec.set(0, 0); // turns off the first bit
bitvec.reset(i); // turns off the ith bit
bitvec.test(0); // returns false because the first bit is off
```

The subscript operator is overloaded on `const`. The `const` version returns a `bool` value `true` if the bit at the given index is on, `false` otherwise. The `nonconst` version returns a special type defined by `bitset` that lets us manipulate the bit value at the given index position:

```
bitvec[0] = 0;           // turn off the bit at position 0
bitvec[31] = bitvec[0]; // set the last bit to the same value as the first bit
bitvec[0].flip();       // flip the value of the bit at position 0
~bitvec[0];             // equivalent operation; flips the bit at position 0
bool b = bitvec[0];     // convert the value of bitvec[0] to bool
```

## Retrieving the Value of a `bitset`

The `to_ulong` and `to_ullong` operations return a value that holds the same bit pattern as the `bitset` object. We can use these operations only if the size of the `bitset` is less than or equal to the corresponding size, `unsigned long` for `to_ulong` and `unsigned long long` for `to_ullong`:

```
unsigned long ulong = bitvec3.to_ulong();
cout << "ulong = " << ulong << endl;
```



These operations throw an `overflow_error` exception (§ 5.6, p. 193) if the value in the `bitset` does not fit in the specified type.

## `bitset` IO Operators

The input operator reads characters from the input stream into a temporary object of type `string`. It reads until it has read as many characters as the size of the corresponding `bitset`, or it encounters a character other than 1 or 0, or it encounters end-of-file or an input error. The `bitset` is then initialized from that temporary `string` (§ 17.2.1, p. 724). If fewer characters are read than the size of the `bitset`, the high-order bits are, as usual, set to 0.

The output operator prints the bit pattern in a `bitset` object:

```
bitset<16> bits;
cin >> bits; // read up to 16 1 or 0 characters from cin
cout << "bits: " << bits << endl; // print what we just read
```

## Using `bitsets`

To illustrate using `bitsets`, we'll reimplement the grading code from § 4.8 (p. 154) that used an `unsigned long` to represent the pass/fail quiz results for 30 students:

```
bool status;
// version using bitwise operators
unsigned long quizA = 0;           // this value is used as a collection of bits
quizA |= 1UL << 27;              // indicate student number 27 passed
status = quizA & (1UL << 27);    // check how student number 27 did
quizA &= ~(1UL << 27);          // student number 27 failed

// equivalent actions using the bitset library
bitset<30> quizB;               // allocate one bit per student; all bits initialized to 0
quizB.set(27);                  // indicate student number 27 passed
status = quizB[27];              // check how student number 27 did
quizB.reset(27);                // student number 27 failed
```

### EXERCISES SECTION 17.2.2

**Exercise 17.10:** Using the sequence 1, 2, 3, 5, 8, 13, 21, initialize a `bitset` that has a 1 bit in each position corresponding to a number in this sequence. Default initialize another `bitset` and write a small program to turn on each of the appropriate bits.

**Exercise 17.11:** Define a data structure that contains an integral object to track responses to a true/false quiz containing 10 questions. What changes, if any, would you need to make in your data structure if the quiz had 100 questions?

**Exercise 17.12:** Using the data structure from the previous question, write a function that takes a question number and a value to indicate a true/false answer and updates the quiz results accordingly.

**Exercise 17.13:** Write an integral object that contains the correct answers for the true/false quiz. Use it to generate grades on the quiz for the data structure from the previous two exercises.

## 17.3 Regular Expressions

A **regular expression** is a way of describing a sequence of characters. Regular expressions are a stunningly powerful computational device. However, describing the languages used to define regular expressions is well beyond the scope of this Primer. Instead, we'll focus on how to use the C++ regular-expression library (RE library), which is part of the new library. The RE library, which is defined in the `regex` header, involves several components, listed in Table 17.4.



If you are not already familiar with using regular expressions, you might want to skim this section to get an idea of the kinds of things regular expressions can do.

**Table 17.4: Regular Expression Library Components**

|                              |                                                                                                               |
|------------------------------|---------------------------------------------------------------------------------------------------------------|
| <code>regex</code>           | Class that represents a regular expression                                                                    |
| <code>regex_match</code>     | Matches a sequence of characters against a regular expression                                                 |
| <code>regex_search</code>    | Finds the first subsequence that matches the regular expression                                               |
| <code>regex_replace</code>   | Replaces a regular expression using a given format                                                            |
| <code>sregex_iterator</code> | Iterator adaptor that calls <code>regex_search</code> to iterate through the matches in a <code>string</code> |
| <code>smatch</code>          | Container class that holds the results of searching a <code>string</code>                                     |
| <code>ssub_match</code>      | Results for a matched subexpression in a <code>string</code>                                                  |

The `regex` class represents a regular expression. Aside from initialization and assignment, `regex` has few operations. The operations on `regex` are listed in Table 17.6 (p. 731).

The functions `regex_match` and `regex_search` determine whether a given character sequence matches a given `regex`. The `regex_match` function returns `true` if the entire input sequence matches the expression; `regex_search` returns `true` if there is a substring in the input sequence that matches. There is also a `regex_replace` function that we'll describe in § 17.3.4 (p. 741).

The arguments to the `regex` functions are described in Table 17.5 (overleaf). These functions return a `bool` and are overloaded: One version takes an additional argument of type `smatch`. If present, these functions store additional information about a successful match in the given `smatch` object.

### 17.3.1 Using the Regular Expression Library

As a fairly simple example, we'll look for words that violate a well-known spelling rule of thumb, "*i* before *e* except after *c*":

```
// find the characters ei that follow a character other than c
string pattern("[^c]ei");
// we want the whole word in which our pattern appears
pattern = "[[:alpha:]]*" + pattern + "[[:alpha:]]*";
regex r(pattern); // construct a regex to find pattern
smatch results; // define an object to hold the results of a search

// define a string that has text that does and doesn't match pattern
string test_str = "receipt freind theif receive";
// use r to find a match to pattern in test_str
if (regex_search(test_str, results, r)) // if there is a match
    cout << results.str() << endl; // print the matching word
```

We start by defining a `string` to hold the regular expression we want to find. The regular expression `[^c]` says we want any character that is not a '`c`', and `[^c]ei` says we want any such letter that is followed by the letters `ei`. This pattern describes strings containing exactly three characters. We want the entire word that contains this pattern. To match the word, we need a regular expression that will match the letters that come before and after our three-letter pattern.

**Table 17.5: Arguments to `regex_search` and `regex_match`**

**Note:** These operations return `bool` indicating whether a match was found.

- (`seq, m, r, mft`) Look for the regular expression in the `regex` object `r` in the character sequence `seq`. `seq` can be a `string`, a pair of iterators denoting a range, or a pointer to a null-terminated character array.  
`m` is a *match* object, which is used to hold details about the match.  
`m` and `seq` must have compatible types (see § 17.3.1 (p. 733)).  
`mft` is an optional `regex_constants::match_flag_type` value.  
These values, listed in Table 17.13 (p. 744), affect the match process.

That regular expression consists of zero or more letters followed by our original three-letter pattern followed by zero or more additional characters. By default, the regular-expression language used by `regex` objects is ECMAScript. In ECMAScript, the pattern `[[:alpha:]]` matches any alphabetic character, and the symbols `+` and `*` signify that we want “one or more” or “zero or more” matches, respectively. Thus, `[[:alpha:]]*` will match zero or more characters.

Having stored our regular expression in `pattern`, we use it to initialize a `regex` object named `r`. We next define a `string` that we’ll use to test our regular expression. We initialize `test_str` with words that match our pattern (e.g., “freind” and “theif”) and words (e.g., “receipt” and “receive”) that don’t. We also define an `smatch` object named `results`, which we will pass to `regex_search`. If a match is found, `results` will hold the details about where the match occurred.

Next we call `regex_search`. If `regex_search` finds a match, it returns `true`. We use the `str` member of `results` to print the part of `test_str` that matched our pattern. The `regex_search` function stops looking as soon as it finds a matching substring in the input sequence. Thus, the output will be

```
freind
```

§ 17.3.2 (p. 734) will show how to find all the matches in the input.

## Specifying Options for a `regex` Object

When we define a `regex` or call `assign` on a `regex` to give it a new value, we can specify one or more flags that affect how the `regex` operates. These flags control the processing done by that object. The last six flags listed in Table 17.6 indicate the language in which the regular expression is written. Exactly one of the flags that specify a language must be set. By default, the `ECMAScript` flag is set, which causes the `regex` to use the ECMA-262 specification, which is the regular expression language that many Web browsers use.

The other three flags let us specify language-independent aspects of the regular-expression processing. For example, we can indicate that we want the regular expression to be matched in a case-independent manner.

As one example, we can use the `icase` flag to find file names that have a particular file extension. Most operating systems recognize extensions in a case-independent manner—we can store a C++ program in a file that ends in `.cc`, or

**Table 17.6: `regex` (and `wregex`) Operations**

|                               |                                                                                                                                                                                                                                                                                                                                                                                                         |
|-------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>regex r(re)</code>      | <i>re</i> represents a regular expression and can be a string, a pair of iterators denoting a range of characters, a pointer to a null-terminated character array, a character pointer and a count, or a braced list of characters. <i>f</i> are flags that specify how the object will execute. <i>f</i> is set from the values listed below. If <i>f</i> is not specified, it defaults to ECMAScript. |
| <code>r1 = re</code>          | Replace the regular expression in <i>r1</i> with <i>re</i> . <i>re</i> represents a regular expression and can be another <code>regex</code> , a string, a pointer to a null-terminated character array, or a braced list of characters.                                                                                                                                                                |
| <code>r1.assign(re, f)</code> | Same effect as using the assignment operator (=). <i>re</i> and optional flag <i>f</i> same as corresponding arguments to <code>regex</code> constructors.                                                                                                                                                                                                                                              |
| <code>r.mark_count()</code>   | Number of subexpressions (which we'll cover in § 17.3.3 (p. 738)) in <i>r</i> .                                                                                                                                                                                                                                                                                                                         |
| <code>r.flags()</code>        | Returns the flags set for <i>r</i> .                                                                                                                                                                                                                                                                                                                                                                    |

*Note: Constructors and assignment operations may throw exceptions of type `regex_error`.*

#### Flags Specified When a `regex` Is Defined

##### Defined in `regex` and `regex_constants::syntax_option_type`

|                         |                                                                     |
|-------------------------|---------------------------------------------------------------------|
| <code>icase</code>      | Ignore case during the match                                        |
| <code>nosubs</code>     | Don't store subexpression matches                                   |
| <code>optimize</code>   | Favor speed of execution over speed of construction                 |
| <code>ECMAScript</code> | Use grammar as specified by ECMA-262                                |
| <code>basic</code>      | Use POSIX basic regular-expression grammar                          |
| <code>extended</code>   | Use POSIX extended regular-expression grammar                       |
| <code>awk</code>        | Use grammar from the POSIX version of the <code>awk</code> language |
| <code>grep</code>       | Use grammar from the POSIX version of <code>grep</code>             |
| <code>egrep</code>      | Use grammar from the POSIX version of <code>egrep</code>            |

.Cc , or .cC , or .CC . We'll write a regular expression to recognize any of these along with other common file extensions as follows:

```
// one or more alphanumeric characters followed by a '.' followed by "cpp" or "cxx" or "cc"
regex r("[[:alnum:]]+\\".(cpp|cxx|cc)$", regex::icase);
smatch results;
string filename;
while (cin >> filename)
    if (regex_search(filename, results, r))
        cout << results.str() << endl; // print the current match
```

This expression will match a string of one or more letters or digits followed by a period and followed by one of three file extensions. The regular expression will match the file extensions regardless of case.

Just as there are special characters in C++ (§ 2.1.3, p. 39), regular-expression languages typically also have special characters. For example, the dot (.) character usually matches any character. As we do in C++, we can escape the special nature of a character by preceding it with a backslash. Because the backslash is also a special character in C++, we must use a second backslash inside a string literal to indicate to C++ that we want a backslash. Hence, we must write \\ . to represent a regular expression that will match a period.

## Errors in Specifying or Using a Regular Expression

We can think of a regular expression as itself a “program” in a simple programming language. That language is not interpreted by the C++ compiler. Instead, a regular expression is “compiled” at run time when a `regex` object is initialized with or assigned a new pattern. As with any programming language, it is possible that the regular expressions we write can have errors.



It is important to realize that the syntactic correctness of a regular expression is evaluated at run time.

If we make a mistake in writing a regular expression, then at *run time* the library will throw an exception (§ 5.6, p. 193) of type `regex_error`. Like the standard exception types, `regex_error` has a `what` operation that describes the error that occurred (§ 5.6.2, p. 195). A `regex_error` also has a member named `code` that returns a numeric code corresponding to the type of error that was encountered. The values `code` returns are implementation defined. The standard errors that the RE library can throw are listed in Table 17.7.

For example, we might inadvertently omit a bracket in a pattern:

```
try {
    // error: missing close bracket afteralnum; the constructor will throw
    regex r("[[:alnum:]]+\\" .(cpp|cxx|cc)$", regex::icase);
} catch (regex_error e)
{ cout << e.what() << "\nerror: " << e.code() << endl; }
```

When run on our system, this program generates

```
regex_error(error_brack):
The expression contained mismatched [ and ].
code: 4
```

**Table 17.7: Regular Expression Error Conditions**

Defined in `regex` and in `regex_constants::error_type`

|                               |                                                                                        |
|-------------------------------|----------------------------------------------------------------------------------------|
| <code>error_collate</code>    | Invalid collating element request                                                      |
| <code>error_ctype</code>      | Invalid character class                                                                |
| <code>error_escape</code>     | Invalid escape character or trailing escape                                            |
| <code>error_backref</code>    | Invalid back reference                                                                 |
| <code>error_brack</code>      | Mismatched bracket ([ or ])                                                            |
| <code>error_paren</code>      | Mismatched parentheses (( or ))                                                        |
| <code>error_brace</code>      | Mismatched brace {{ or }}                                                              |
| <code>error_badbrace</code>   | Invalid range inside a {}                                                              |
| <code>error_range</code>      | Invalid character range (e.g., [z-a])                                                  |
| <code>error_space</code>      | Insufficient memory to handle this regular expression                                  |
| <code>error_badrepeat</code>  | A repetition character (*, ?, +, or {}) was not preceded by a valid regular expression |
| <code>error_complexity</code> | The requested match is too complex                                                     |
| <code>error_stack</code>      | Insufficient memory to evaluate a match                                                |

Our compiler defines the `code` member to return the position of the error as listed in Table 17.7, counting, as usual, from zero.

#### **ADVICE: AVOID CREATING UNNECESSARY REGULAR EXPRESSIONS**

As we've seen, the "program" that a regular expression represents is compiled at run time, not at compile time. Compiling a regular expression can be a surprisingly slow operation, especially if you're using the extended regular-expression grammar or are using complicated expressions. As a result, constructing a `regex` object and assigning a new regular expression to an existing `regex` can be time-consuming. To minimize this overhead, you should try to avoid creating more `regex` objects than needed. In particular, if you use a regular expression in a loop, you should create it outside the loop rather than recompiling it on each iteration.

## **Regular Expression Classes and the Input Sequence Type**

We can search any of several types of input sequence. The input can be ordinary `char` data or `wchar_t` data and those characters can be stored in a library `string` or in an array of `char` (or the wide character versions, `wstring` or array of `wchar_t`). The RE library defines separate types that correspond to these differing types of input sequences.

For example, the `regex` class holds regular expressions of type `char`. The library also defines a `wregex` class that holds type `wchar_t` and has all the same operations as `regex`. The only difference is that the initializers of a `wregex` must use `wchar_t` instead of `char`.

The match and iterator types (which we will cover in the following sections) are more specific. These types differ not only by the character type, but also by whether the sequence is in a library `string` or an array: `smatch` represents `string` input sequences; `cmatch`, character array sequences; `wsmatch`, wide string (`wstring`) input; and `wcmatch`, arrays of wide characters.

**Table 17.8: Regular Expression Library Classes**

| If Input Sequence Has Type  | Use Regular Expression Classes                                                                            |
|-----------------------------|-----------------------------------------------------------------------------------------------------------|
| <code>string</code>         | <code>regex</code> , <code>smatch</code> , <code>ssub_match</code> , and <code>sregex_iterator</code>     |
| <code>const char*</code>    | <code>regex</code> , <code>cmatch</code> , <code>csub_match</code> , and <code>cregex_iterator</code>     |
| <code>wstring</code>        | <code>wregex</code> , <code>wsmatch</code> , <code>wssub_match</code> , and <code>wsregex_iterator</code> |
| <code>const wchar_t*</code> | <code>wregex</code> , <code>wcmatch</code> , <code>wcsub_match</code> , and <code>wcregex_iterator</code> |

The important point is that the RE library types we use must match the type of the input sequence. Table 17.8 indicates which types correspond to which kinds of input sequences. For example:

```
regex r("[[:alnum:]]+\.\(cpp|cxx|cc)\$", regex::icase);
smatch results; // will match a string input sequence, but not char*
if (regex_search("myfile.cc", results, r)) // error: char* input
    cout << results.str() << endl;
```

The (C++) compiler will reject this code because the type of the match argument and the type of the input sequence do not match. If we want to search a character array, then we must use a `cmatch` object:

```
cmatch results; // will match character array input sequences
if (regex_search("myfile.cc", results, rx))
    cout << results.str() << endl; // print the current match
```

In general, our programs will use `string` input sequences and the corresponding `string` versions of the RE library components.

### EXERCISES SECTION 17.3.1

**Exercise 17.14:** Write several regular expressions designed to trigger various errors. Run your program to see what output your compiler generates for each error.

**Exercise 17.15:** Write a program using the pattern that finds words that violate the “*i* before *e* except after *c*” rule. Have your program prompt the user to supply a word and indicate whether the word is okay or not. Test your program with words that do and do not violate the rule.

**Exercise 17.16:** What would happen if your `regex` object in the previous program were initialized with “[*c*]ei”? Test your program using that pattern to see whether your expectations were correct.

## 17.3.2 The Match and Regex Iterator Types

The program on page 729 that found violations of the “*i* before *e* except after *c*” grammar rule printed only the first match in its input sequence. We can get all the matches by using an `sregex_iterator`. The regex iterators are iterator adaptors (§ 9.6, p. 368) that are bound to an input sequence and a `regex` object. As described in Table 17.8 (on the previous page), there are specific regex iterator types that correspond to each of the different types of input sequences. The iterator operations are described in Table 17.9 (p. 736).

When we bind an `sregex_iterator` to a `string` and a `regex` object, the iterator is automatically positioned on the first match in the given `string`. That is, the `sregex_iterator` constructor calls `regex_search` on the given `string` and `regex`. When we dereference the iterator, we get an `smatch` object corresponding to the results from the most recent search. When we increment the iterator, it calls `regex_search` to find the next match in the input `string`.

### Using an `sregex_iterator`

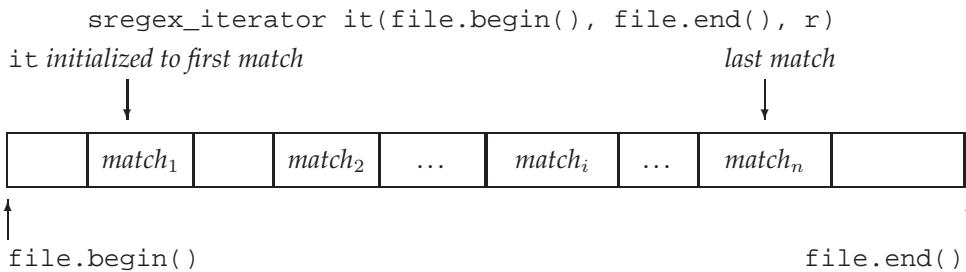
As an example, we’ll extend our program to find all the violations of the “*i* before *e* except after *c*” grammar rule in a file of text. We’ll assume that the `string` named `file` holds the entire contents of the input file that we want to search. This version of the program will use the same pattern as our original one, but will use a `sregex_iterator` to do the search:

```
// find the characters ei that follow a character other than c
string pattern("[^c]ei");
// we want the whole word in which our pattern appears
pattern = "[[:alpha:]]*" + pattern + "[[:alpha:]]*";
regex r(pattern, regex::icase); // we'll ignore case in doing the match
// it will repeatedly call regex_search to find all matches in file
for (sregex_iterator it(file.begin(), file.end(), r), end_it;
     it != end_it; ++it)
    cout << it->str() << endl; // matched word
```

The for loop iterates through each match to `r` inside `file`. The initializer in the for defines `it` and `end_it`. When we define `it`, the `sregex_iterator` constructor calls `regex_search` to position `it` on the first match in `file`. The empty `sregex_iterator`, `end_it`, acts as the off-the-end iterator. The increment in the for “advances” the iterator by calling `regex_search`. When we dereference the iterator, we get an `smatch` object representing the current match. We call the `str` member of the match to print the matching word.

We can think of this loop as jumping from match to match as illustrated in Figure 17.1.

**Figure 17.1: Using an `sregex_iterator`**



## Using the Match Data

If we run this loop on `test_str` from our original program, the output would be

```
freind
theif
```

However, finding just the words that match our expression is not so useful. If we ran the program on a larger input sequence—for example, on the text of this chapter—we’d want to see the context within which the word occurs, such as

```
hey read or write according to the type
    >>> being <<
handled. The input operators ignore whi
```

In addition to letting us print the part of the input string that was matched, the match classes give us more detailed information about the match. The operations on these types are listed in Table 17.10 (p. 737) and Table 17.11 (p. 741).

**Table 17.9: `sregex_iterator` Operations**

**These operations also apply to `cregex_iterator`,  
`wsregex_iterator`, and `wcregex_iterator`**

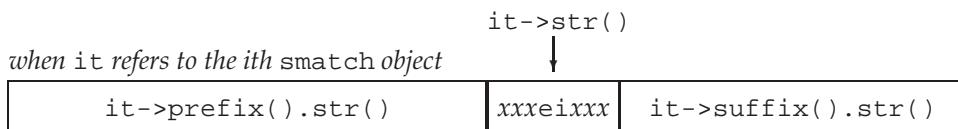
|                                                        |                                                                                                                                                                                                 |
|--------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>sregex_iterator</code> <code>it(b, e, r);</code> | it is an <code>sregex_iterator</code> that iterates through the string denoted by iterators b and e. Calls <code>regex_search(b, e, r)</code> to position it on the first match in the input.   |
| <code>sregex_iterator</code> <code>end;</code>         | Off-the-end iterator for <code>sregex_iterator</code> .                                                                                                                                         |
| <code>*it</code>                                       | Returns a reference to the <code>smatch</code> object or a pointer to the <code>smatch</code> object from the most recent call to <code>regex_search</code> .                                   |
| <code>it-&gt;</code>                                   | Calls <code>regex_search</code> on the input sequence starting just after the current match. The prefix version returns a reference to the incremented iterator; postfix returns the old value. |
| <code>it++</code>                                      |                                                                                                                                                                                                 |
| <code>it1 == it2</code>                                | Two <code>sregex_iterator</code> s are equal if they are both the off-the-end iterator.                                                                                                         |
| <code>it1 != it2</code>                                | Two non-end iterators are equal if they are constructed from the same input sequence and <code>regex</code> object.                                                                             |

We'll have more to say about the `smatch` and `ssub_match` types in the next section. For now, what we need to know is that these types let us see the context of a match. The match types have members named `prefix` and `suffix`, which return a `ssub_match` object representing the part of the input sequence ahead of and after the current match, respectively. A `ssub_match` object has members named `str` and `length`, which return the matched string and size of that string, respectively. We can use these operations to rewrite the loop of our grammar program:

```
// same for loop header as before
for (sregex_iterator it(file.begin(), file.end(), r), end_it;
     it != end_it; ++it) {
    auto pos = it->prefix().length(); // size of the prefix
    pos = pos > 40 ? pos - 40 : 0; // we want up to 40 characters
    cout << it->prefix().str().substr(pos) // last part of the prefix
        << "\n\t\t>>" << it->str() << " <<<\n" // matched word
        << it->suffix().str().substr(0, 40) // first part of the suffix
        << endl;
}
```

The loop itself operates the same way as our previous program. What's changed is the processing inside the `for`, which is illustrated in Figure 17.2.

We call `prefix`, which returns an `ssub_match` object that represents the part of `file` ahead of the current match. We call `length` on that `ssub_match` to find out how many characters are in the part of `file` ahead of the match. Next we adjust `pos` to be the index of the character 40 from the end of the prefix. If the prefix has fewer than 40 characters, we set `pos` to 0, which means we'll print the entire prefix. We use `substr` (§ 9.5.1, p. 361) to print from the given position to the end of the prefix.

**Figure 17.2: The `smatch` Object Representing a Particular Match**

Having printed the characters that precede the match, we next print the match itself with some additional formatting so that the matched word will stand out in the output. After printing the matched portion, we print (up to) the first 40 characters in the part of `file` that comes after this match.

**Table 17.10: `smatch` Operations**

These operations also apply to the `cmatch`, `wsmatch`, `wcmatch` and the corresponding `csub_match`, `wssub_match`, and `wcsub_match` types.

|                            |                                                                                                                                                                                                           |
|----------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>m.ready()</code>     | true if <code>m</code> has been set by a call to <code>regex_search</code> or <code>regex_match</code> ; false otherwise. Operations on <code>m</code> are undefined if <code>ready</code> returns false. |
| <code>m.size()</code>      | Zero if the match failed; otherwise, one plus the number of subexpressions in the most recently matched regular expression.                                                                               |
| <code>m.empty()</code>     | true if <code>m.size()</code> is zero.                                                                                                                                                                    |
| <code>m.prefix()</code>    | An <code>ssub_match</code> representing the sequence before the match.                                                                                                                                    |
| <code>m.suffix()</code>    | An <code>ssub_match</code> representing the part after the end of the match.                                                                                                                              |
| <code>m.format(...)</code> | See Table 17.12 (p. 742).                                                                                                                                                                                 |

In the operations that take an index, `n` defaults to zero and must be less than `m.size()`.

The first submatch (the one with index 0) represents the overall match.

|                                               |                                                                                                                                                                                                                    |
|-----------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>m.length(n)</code>                      | Size of the <code>n</code> th matched subexpression.                                                                                                                                                               |
| <code>m.position(n)</code>                    | Distance of the <code>n</code> th subexpression from the start of the sequence.                                                                                                                                    |
| <code>m.str(n)</code>                         | The matched string for the <code>n</code> th subexpression.                                                                                                                                                        |
| <code>m[n]</code>                             | <code>ssub_match</code> object corresponding to the <code>n</code> th subexpression.                                                                                                                               |
| <code>m.begin()</code> , <code>m.end()</code> | Iterators across the <code>sub_match</code> elements in <code>m</code> . As usual, <code>cbegin</code> , <code>cbegin()</code> , <code>m.cend()</code> and <code>cend</code> return <code>const_iterators</code> . |

## EXERCISES SECTION 17.3.2

**Exercise 17.17:** Update your program so that it finds all the words in an input sequence that violate the “ei” grammar rule.

**Exercise 17.18:** Revise your program to ignore words that contain “ei” but are not misspellings, such as “albeit” and “neighbor.”

### 17.3.3 Using Subexpressions

A pattern in a regular expression often contains one or more **subexpressions**. A subexpression is a part of the pattern that itself has meaning. Regular-expression grammars typically use parentheses to denote subexpressions.

As an example, the pattern that we used to match C++ files (§ 17.3.1, p. 730) used parentheses to group the possible file extensions. Whenever we group alternatives using parentheses, we are also declaring that those alternatives form a subexpression. We can rewrite that expression so that it gives us access to the file name, which is the part of the pattern that precedes the period, as follows:

```
// r has two subexpressions: the first is the part of the file name before the period
// the second is the file extension
regex r("([[:alnum:]]+)\\.\\.(cpp|cxx|cc)$", regex::icase);
```

Our pattern now has two parenthesized subexpressions:

- `([[:alnum:]]+)`, which is a sequence of one or more characters
- `(cpp|cxx|cc)`, which is the file extension

We can also rewrite the program from § 17.3.1 (p. 730) to print just the file name by changing the output statement:

```
if (regex_search(filename, results, r))
    cout << results.str(1) << endl; // print the first subexpression
```

As in our original program, we call `regex_search` to look for our pattern `r` in the string named `filename`, and we pass the `smatch` object `results` to hold the results of the match. If the call succeeds, then we print the results. However, in this program, we print `str(1)`, which is the match for the first subexpression.

In addition to providing information about the overall match, the match objects provide access to each matched subexpression in the pattern. The submatches are accessed positionally. The first submatch, which is at position 0, represents the match for the entire pattern. Each subexpression appears in order thereafter. Hence, the file name, which is the first subexpression in our pattern, is at position 1, and the file extension is in position 2.

For example, if the file name is `foo.cpp`, then `results.str(0)` will hold `foo.cpp`; `results.str(1)` will be `foo`; and `results.str(2)` will be `cpp`. In this program, we want the part of the name before the period, which is the first subexpression, so we print `results.str(1)`.

### Subexpressions for Data Validation

One common use for subexpressions is to validate data that must match a specific format. For example, U.S. phone numbers have ten digits, consisting of an area code and a seven-digit local number. The area code is often, but not always, enclosed in parentheses. The remaining seven digits can be separated by a dash, a dot, or a space; or not separated at all. We might want to allow data with any of these formats and reject numbers in other forms. We'll do a two-step process: First,

we'll use a regular expression to find sequences that might be phone numbers and then we'll call a function to complete the validation of the data.

Before we write our phone number pattern, we need to describe a few more aspects of the ECMAScript regular-expression language:

- $\backslash\{d\}$  represents a single digit and  $\backslash\{d\}\{n\}$  represents a sequence of  $n$  digits. (E.g.,  $\backslash\{d\}\{3\}$  matches a sequence of three digits.)
- A collection of characters inside square brackets allows a match to any of those characters. (E.g.,  $[-\cdot]$  matches a dash, a dot, or a space. Note that a dot has no special meaning inside brackets.)
- A component followed by '?' is optional. (E.g.,  $\backslash\{d\}\{3\}[-\cdot]?\backslash\{d\}\{4\}$  matches three digits followed by an optional dash, period, or space, followed by four more digits. This pattern would match 555-0132 or 555.0132 or 555 0132 or 5550132.)
- Like C++, ECMAScript uses a backslash to indicate that a character should represent itself, rather than its special meaning. Because our pattern includes parentheses, which are special characters in ECMAScript, we must represent the parentheses that are part of our pattern as  $\backslash($  or  $\backslash)$ .

Because backslash is a special character in C++, each place that a  $\backslash$  appears in the pattern, we must use a second backslash to indicate to C++ that we want a backslash. Hence, we write  $\backslash\backslash\{d\}\{3\}$  to represent the regular expression  $\backslash\{d\}\{3\}$ .

In order to validate our phone numbers, we'll need access to the components of the pattern. For example, we'll want to verify that if a number uses an opening parenthesis for the area code, it also uses a close parenthesis after the area code. That is, we'd like to reject a number such as (908.555.1800.

To get at the components of the match, we need to define our regular expression using subexpressions. Each subexpression is marked by a pair of parentheses:

```
// our overall expression has seven subexpressions: (ddd) separator ddd separator dddd
// subexpressions 1, 3, 4, and 6 are optional; 2, 5, and 7 hold the number
"(\backslash\()?(\\d\{3\})(\\))?([-\.\.])?(\\d\{3\})([-\.\.])?(\\d\{4\})";
```

Because our pattern uses parentheses, and because we must escape backslashes, this pattern can be hard to read (and write!). The easiest way to read it is to pick off each (parenthesized) subexpression:

1.  $(\backslash\()?$  an optional open parenthesis for the area code
2.  $(\\d\{3\})$  the area code
3.  $(\\))?$  an optional close parenthesis for the area code
4.  $([-\.\.])?$  an optional separator after the area code
5.  $(\\d\{3\})$  the next three digits of the number
6.  $([-\.\.])?$  another optional separator
7.  $(\\d\{4\})$  the final four digits of the number

The following code uses this pattern to read a file and find data that match our overall phone pattern. It will call a function named `valid` to check whether the number has a valid format:

```
string phone =
    "(\\()?(\\d{3})(\\))?([- . ])?(\\d{3})([- . ]?)(\\d{4})";
regex r(phone); // a regex to find our pattern
smatch m;
string s;
// read each record from the input file
while (getline(cin, s)) {
    // for each matching phone number
    for (sregex_iterator it(s.begin(), s.end(), r), end_it;
         it != end_it; ++it)
        // check whether the number's formatting is valid
        if (valid(*it))
            cout << "valid: " << it->str() << endl;
        else
            cout << "not valid: " << it->str() << endl;
}
```

## Using the Submatch Operations

We'll use submatch operations, which are outlined in Table 17.11, to write the `valid` function. It is important to keep in mind that our pattern has seven subexpressions. As a result, each `smatch` object will contain eight `ssub_match` elements. The element at `[0]` represents the overall match; the elements `[1]...[7]` represent each of the corresponding subexpressions.

When we call `valid`, we know that we have an overall match, but we do not know which of our optional subexpressions were part of that match. The `matched` member of the `ssub_match` corresponding to a particular subexpression is `true` if that subexpression is part of the overall match.

In a valid phone number, the area code is either fully parenthesized or not parenthesized at all. Therefore, the work `valid` does depends on whether the number starts with a parenthesis or not:

```
bool valid(const smatch& m)
{
    // if there is an open parenthesis before the area code
    if (m[1].matched)
        // the area code must be followed by a close parenthesis
        // and followed immediately by the rest of the number or a space
        return m[3].matched
            && (m[4].matched == 0 || m[4].str() == " ");
    else
        // then there can't be a close after the area code
        // the delimiters between the other two components must match
        return !m[3].matched
            && m[4].str() == m[6].str();
}
```

**Table 17.11: Submatch Operations**

|                                                                                                                                             |                                                                                                                                                                                                                  |
|---------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>Note: These operations apply to <code>ssub_match</code>, <code>csub_match</code>, <code>wssub_match</code>, <code>wcsub_match</code></i> |                                                                                                                                                                                                                  |
| <code>matched</code>                                                                                                                        | A public <code>bool</code> data member that indicates whether this <code>ssub_match</code> was matched.                                                                                                          |
| <code>first</code>                                                                                                                          | public data members that are iterators to the start and one past the end of the matching sequence. If there was no match, then <code>first</code> and <code>second</code> are equal.                             |
| <code>second</code>                                                                                                                         |                                                                                                                                                                                                                  |
| <code>length()</code>                                                                                                                       | The size of this match. Returns 0 if <code>matched</code> is <code>false</code> .                                                                                                                                |
| <code>str()</code>                                                                                                                          | Returns a string containing the matched portion of the input. Returns the empty string if <code>matched</code> is <code>false</code> .                                                                           |
| <code>s = ssub</code>                                                                                                                       | Convert the <code>ssub_match</code> object <code>ssub</code> to the <code>string</code> <code>s</code> . Equivalent to <code>s = ssub.str()</code> . The conversion operator is not explicit (§ 14.9.1, p. 581). |

We start by checking whether the first subexpression (i.e., the open parenthesis) matched. That subexpression is in `m[1]`. If it matched, then the number starts with an open parenthesis. In this case, the overall number is valid if the subexpression following the area code also matched (meaning that there was a close parenthesis after the area code). Moreover, if the number is correctly parenthesized, then the next character must be a space or the first digit in the next part of the number.

If `m[1]` didn't match (i.e., there was no open parenthesis), the subexpression following the area code must also be empty. If it's empty, then the number is valid if the remaining separators are equal and not otherwise.

### EXERCISES SECTION 17.3.3

**Exercise 17.19:** Why is it okay to call `m[4].str()` without first checking whether `m[4]` was matched?

**Exercise 17.20:** Write your own version of the program to validate phone numbers.

**Exercise 17.21:** Rewrite your phone number program from § 8.3.2 (p. 323) to use the `valid` function defined in this section.

**Exercise 17.22:** Rewrite your phone program so that it allows any number of whitespace characters to separate the three parts of a phone number.

**Exercise 17.23:** Write a regular expression to find zip codes. A zip code can have five or nine digits. The first five digits can be separated from the remaining four by a dash.

### 17.3.4 Using `regex_replace`

Regular expressions are often used when we need not only to find a given sequence but also to replace that sequence with another one. For example, we might want to translate U.S. phone numbers into the form "ddd.ddd.dddd," where the area code and next three digits are separated by a dot.

When we want to find and replace a regular expression in the input sequence, we call `regex_replace`. Like the search functions, `regex_replace`, which is described in Table 17.12, takes an input character sequence and a `regex` object. We must also pass a string that describes the output we want.

We compose a replacement string by including the characters we want, intermixed with subexpressions from the matched substring. In this case, we want to use the second, fifth, and seventh subexpressions in our replacement string. We'll ignore the first, third, fourth, and sixth, because these were used in the original formatting of the number but are not part of our replacement format. We refer to a particular subexpression by using a \$ symbol followed by the index number for a subexpression:

```
string fmt = "$2.$5.$7"; // reformat numbers to dddddd.ddd
```

We can use our regular-expression pattern and the replacement string as follows:

```
regex r(phone); // a regex to find our pattern
string number = "(908) 555-0132";
cout << regex_replace(number, r, fmt) << endl;
```

The output from this program is

```
908.555.0132
```

**Table 17.12: Regular Expression Replace Operations**

|                                                                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|---------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>m.format(dest, fmt, mft)</code>                               | Produces formatted output using the format string <code>fmt</code> , the match in <code>m</code> , and the optional <code>match_flag_type</code> flags in <code>mft</code> . The first version writes to the output iterator <code>dest</code> (§ 10.5.1, p. 410) and takes <code>fmt</code> that is either a <code>string</code> or a pair of pointers denoting a range in a character array. The second version returns a <code>string</code> that holds the output and takes <code>fmt</code> that is a <code>string</code> or a pointer to a null-terminated character array. <code>mft</code> defaults to <code>format_default</code> .                                                                                                              |
| <code>regex_replace</code><br><code>(dest, seq, r, fmt, mft)</code> | Iterates through <code>seq</code> , using <code>regex_search</code> to find successive matches to <code>regex r</code> . Uses the format string <code>fmt</code> and optional <code>match_flag_type</code> flags in <code>mft</code> to produce its output. The first version writes to the output iterator <code>dest</code> , and takes a pair of iterators to denote <code>seq</code> . The second returns a <code>string</code> that holds the output and <code>seq</code> can be either a <code>string</code> or a pointer to a null-terminated character array. In all cases, <code>fmt</code> can be either a <code>string</code> or a pointer to a null-terminated character array, and <code>mft</code> defaults to <code>match_default</code> . |
| <code>regex_replace</code><br><code>(seq, r, fmt, mft)</code>       |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |

## Replacing Only Part of the Input Sequence

A more interesting use of our regular-expression processing would be to replace phone numbers that are embedded in a larger file. For example, we might have a file of names and phone number that had data like this:

```
morgan (201) 555-0168 862-555-0123
drew (973)555.0130
lee (609) 555-0132 2015550175 800.555-0100
```

that we want to transform to data like this:

```
morgan 201.555.0168 862.555.0123
drew 973.555.0130
lee 609.555.0132 201.555.0175 800.555.0100
```

We can generate this transformation with the following program:

```
int main()
{
    string phone =
        "(\\()?(\\d{3})(\\))?([- .])?(\\d{3})([- .])?(\\d{4})";
    regex r(phone); // a regex to find our pattern
    smatch m;
    string s;
    string fmt = "$2.$5.$7"; // reformat numbers to ddd.ddd.dddd
    // read each record from the input file
    while (getline(cin, s))
        cout << regex_replace(s, r, fmt) << endl;
    return 0;
}
```

We read each record into `s` and hand that record to `regex_replace`. This function finds and transforms *all* the matches in its input sequence.

## Flags to Control Matches and Formatting

Just as the library defines flags to direct how to process a regular expression, the library also defines flags that we can use to control the match process or the formatting done during a replacement. These values are listed in Table 17.13 (overleaf). These flags can be passed to the `regex_search` or `regex_match` functions or to the `format` members of class `smatch`.

The match and format flags have type `match_flag_type`. These values are defined in a namespace named `regex_constants`. Like `placeholders`, which we used with `bind` (§ 10.3.4, p. 399), `regex_constants` is a namespace defined inside the `std` namespace. To use a name from `regex_constants`, we must qualify that name with the names of both namespaces:

```
using std::regex_constants::format_no_copy;
```

This declaration says that when our code uses `format_no_copy`, we want the object of that name from the namespace `std::regex_constants`. We can instead provide the alternative form of using that we will cover in § 18.2.2 (p. 792):

```
using namespace std::regex_constants;
```

**Table 17.13: Match Flags**

| Defined in <code>regex_constants::match_flag_type</code> |                                                              |
|----------------------------------------------------------|--------------------------------------------------------------|
| <code>match_default</code>                               | Equivalent to <code>format_default</code>                    |
| <code>match_not_bol</code>                               | Don't treat the first character as the beginning of the line |
| <code>match_not_eol</code>                               | Don't treat the last character as the end of the line        |
| <code>match_not_bow</code>                               | Don't treat the first character as the beginning of a word   |
| <code>match_not_eow</code>                               | Don't treat the last character as the end of a word          |
| <code>match_any</code>                                   | If there is more than one match, any match can be returned   |
| <code>match_not_null</code>                              | Don't match an empty sequence                                |
| <code>match_continuous</code>                            | The match must begin with the first character in the input   |
| <code>match_prev_avail</code>                            | The input sequence has characters before the first           |
| <code>format_default</code>                              | Replacement string uses the ECMAScript rules                 |
| <code>format_sed</code>                                  | Replacement string uses the rules from POSIX sed             |
| <code>format_no_copy</code>                              | Don't output the unmatched parts of the input                |
| <code>format_first_only</code>                           | Replace only the first occurrence                            |

## Using Format Flags

By default, `regex_replace` outputs its entire input sequence. The parts that don't match the regular expression are output without change; the parts that do match are formatted as indicated by the given format string. We can change this default behavior by specifying `format_no_copy` in the call to `regex_replace`:

```
// generate just the phone numbers: use a new format string
string fmt2 = "$2.$5.$7 " ; // put space after the last number as a separator
// tell regex_replace to copy only the text that it replaces
cout << regex_replace(s, r, fmt2, format_no_copy) << endl;
```

Given the same input, this version of the program generates

```
201.555.0168 862.555.0123
973.555.0130
609.555.0132 201.555.0175 800.555.0100
```

### EXERCISES SECTION 17.3.4

**Exercise 17.24:** Write your own version of the program to reformat phone numbers.

**Exercise 17.25:** Rewrite your phone program so that it writes only the first phone number for each person.

**Exercise 17.26:** Rewrite your phone program so that it writes only the second and subsequent phone numbers for people with more than one phone number.

**Exercise 17.27:** Write a program that reformats a nine-digit zip code as dddd-dddd.

## 17.4 Random Numbers

Programs often need a source of random numbers. Prior to the new standard, both C and C++ relied on a simple C library function named `rand`. That function produces pseudorandom integers that are uniformly distributed in the range from 0 to a system-dependent maximum value that is at least 32767.

C++  
11

The `rand` function has several problems: Many, if not most, programs need random numbers in a different range from the one produced by `rand`. Some applications require random floating-point numbers. Some programs need numbers that reflect a nonuniform distribution. Programmers often introduce nonrandomness when they try to transform the range, type, or distribution of the numbers generated by `rand`.

The random-number library, defined in the `random` header, solves these problems through a set of cooperating classes: **random-number engines** and **random-number distribution classes**. These classes are described in Table 17.14. An engine generates a sequence of unsigned random numbers. A distribution uses an engine to generate random numbers of a specified type, in a given range, distributed according to a particular probability distribution.



C++ programs should not use the library `rand` function. Instead, they should use the `default_random_engine` along with an appropriate distribution object.

**Table 17.14: Random Number Library Components**

|              |                                                                                               |
|--------------|-----------------------------------------------------------------------------------------------|
| Engine       | Types that generate a sequence of random unsigned integers                                    |
| Distribution | Types that use an engine to return numbers according to a particular probability distribution |

### 17.4.1 Random-Number Engines and Distribution

The random-number engines are function-object classes (§ 14.8, p. 571) that define a call operator that takes no arguments and returns a random `unsigned` number. We can generate raw random numbers by calling an object of a random-number engine type:

```
default_random_engine e; // generates random unsigned integers
for (size_t i = 0; i < 10; ++i)
    // e() "calls" the object to produce the next random number
    cout << e() << " ";
```

On our system, this program generates:

```
16807 282475249 1622650073 984943658 1144108930 470211272 ...
```

Here, we defined an object named `e` that has type `default_random_engine`. Inside the `for`, we call the object `e` to obtain the next random number.

The library defines several random-number engines that differ in terms of their performance and quality of randomness. Each compiler designates one of these engines as the `default_random_engine` type. This type is intended to be the engine with the most generally useful properties. Table 17.15 lists the engine operations and the engine types defined by the standard are listed in § A.3.2 (p. 884).

For most purposes, the output of an engine is not directly usable, which is why we described them earlier as raw random numbers. The problem is that the numbers usually span a range that differs from the one we need. *Correctly* transforming the range of a random number is surprisingly hard.

## Distribution Types and Engines

To get a number in a specified range, we use an object of a distribution type:

```
// uniformly distributed from 0 to 9 inclusive
uniform_int_distribution<unsigned> u(0, 9);
default_random_engine e; // generates unsigned random integers
for (size_t i = 0; i < 10; ++i)
    // u uses e as a source of numbers
    // each call returns a uniformly distributed value in the specified range
    cout << u(e) << " ";
```

This code produces output such as

```
0 1 7 4 5 2 0 6 6 9
```

Here we define `u` as a `uniform_int_distribution<unsigned>`. That type generates uniformly distributed `unsigned` values. When we define an object of this type, we can supply the minimum and maximum values we want. In this program, `u(0, 9)` says that we want numbers to be in the range 0 to 9 *inclusive*. The random number distributions use inclusive ranges so that we can obtain every possible value of the given integral type.

Like the engine types, the distribution types are also function-object classes. The distribution types define a call operator that takes a random-number engine as its argument. The distribution object uses its engine argument to produce random numbers that the distribution object maps to the specified distribution.

Note that we pass the engine object itself, `u(e)`. Had we written the call as `u(e())`, we would have tried to pass the next value generated by `e` to `u`, which would be a compile-time error. We pass the engine, not the next result of the engine, because some distributions may need to call the engine more than once.



When we refer to a **random-number generator**, we mean the combination of a distribution object with an engine.

## Comparing Random Engines and the `rand` Function

For readers familiar with the C library `rand` function, it is worth noting that the output of calling a `default_random_engine` object is similar to the output of `rand`. Engines deliver `unsigned` integers in a system-defined range. The range

for `rand` is 0 to `RAND_MAX`. The range for an engine type is returned by calling the `min` and `max` members on an object of that type:

```
cout << "min: " << e.min() << " max: " << e.max() << endl;
```

On our system this program produces the following output:

```
min: 1 max: 2147483646
```

**Table 17.15: Random Number Engine Operations**

|                                  |                                                                                                     |
|----------------------------------|-----------------------------------------------------------------------------------------------------|
| <code>Engine e;</code>           | Default constructor; uses the default seed for the engine type                                      |
| <code>Engine e(s);</code>        | Uses the integral value <code>s</code> as the seed                                                  |
| <code>e.seed(s)</code>           | Reset the state of the engine using the seed <code>s</code>                                         |
| <code>e.min()</code>             | The smallest and largest numbers this generator will generate                                       |
| <code>e.max()</code>             |                                                                                                     |
| <code>Engine::result_type</code> | The unsigned integral type this engine generates                                                    |
| <code>e.discard(u)</code>        | Advance the engine by <code>u</code> steps; <code>u</code> has type <code>unsigned long long</code> |

## Engines Generate a Sequence of Numbers

Random number generators have one property that often confuses new users: Even though the numbers that are generated appear to be random, a given generator returns the same sequence of numbers each time it is run. The fact that the sequence is unchanging is very helpful during testing. On the other hand, programs that use random-number generators have to take this fact into account.

As one example, assume we need a function that will generate a `vector` of 100 random integers uniformly distributed in the range from 0 to 9. We might think we'd write this function as follows:

```
// almost surely the wrong way to generate a vector of random integers
// output from this function will be the same 100 numbers on every call!
vector<unsigned> bad_randVec()
{
    default_random_engine e;
    uniform_int_distribution<unsigned> u(0,9);
    vector<unsigned> ret;
    for (size_t i = 0; i < 100; ++i)
        ret.push_back(u(e));
    return ret;
}
```

However, this function will return the same `vector` every time it is called:

```
vector<unsigned> v1(bad_randVec());
vector<unsigned> v2(bad_randVec());
// will print equal
cout << ((v1 == v2) ? "equal" : "not equal") << endl;
```

This code will print `equal` because the vectors `v1` and `v2` have the same values.

The right way to write our function is to make the engine and associated distribution objects `static` (§ 6.1.1, p. 205):

```
// returns a vector of 100 uniformly distributed random numbers
vector<unsigned> good_randVec()
{
    // because engines and distributions retain state, they usually should be
    // defined as static so that new numbers are generated on each call
    static default_random_engine e;
    static uniform_int_distribution<unsigned> u(0,9);
    vector<unsigned> ret;
    for (size_t i = 0; i < 100; ++i)
        ret.push_back(u(e));
    return ret;
}
```

Because `e` and `u` are `static`, they will hold their state across calls to the function. The first call will use the first 100 random numbers from the sequence `u(e)` generates, the second call will get the next 100, and so on.



A given random-number generator always produces the same sequence of numbers. A function with a local random-number generator should make that generator (both the engine and distribution objects) `static`. Otherwise, the function will generate the identical sequence on each call.

## Seeding a Generator

The fact that a generator returns the same sequence of numbers is helpful during debugging. However, once our program is tested, we often want to cause each run of the program to generate different random results. We do so by providing a `seed`. A seed is a value that an engine can use to cause it to start generating numbers at a new point in its sequence.

We can seed an engine in one of two ways: We can provide the seed when we create an engine object, or we can call the engine's `seed` member:

```
default_random_engine e1;           // uses the default seed
default_random_engine e2(2147483646); // use the given seed value

// e3 and e4 will generate the same sequence because they use the same seed
default_random_engine e3;           // uses the default seed value
e3.seed(32767);                   // call seed to set a new seed value
default_random_engine e4(32767);    // set the seed value to 32767
for (size_t i = 0; i != 100; ++i) {
    if (e1() == e2())
        cout << "unseeded match at iteration: " << i << endl;
    if (e3() != e4())
        cout << "seeded differs at iteration: " << i << endl;
}
```

Here we define four engines. The first two, `e1` and `e2`, have different seeds and

should generate different sequences. The second two, `e3` and `e4`, have the same seed value. These two objects *will* generate the same sequence.

Picking a good seed, like most things about generating good random numbers, is surprisingly hard. Perhaps the most common approach is to call the system `time` function. This function, defined in the `ctime` header, returns the number of seconds since a given epoch. The `time` function takes a single parameter that is a pointer to a structure into which to write the time. If that pointer is null, the function just returns the time:

```
default_random_engine e1(time(0)); // a somewhat random seed
```

Because `time` returns time as the number of seconds, this seed is useful only for applications that generate the seed at second-level, or longer, intervals.



**WARNING** Using `time` as a seed usually doesn't work if the program is run repeatedly as part of an automated process; it might wind up with the same seed several times.

### EXERCISES SECTION 17.4.1

**Exercise 17.28:** Write a function that generates and returns a uniformly distributed random `unsigned int` each time it is called.

**Exercise 17.29:** Allow the user to supply a seed as an optional argument to the function you wrote in the previous exercise.

**Exercise 17.30:** Revise your function again this time to take a minimum and maximum value for the numbers that the function should return.

## 17.4.2 Other Kinds of Distributions

The engines produce `unsigned` numbers, and each number in the engine's range has the same likelihood of being generated. Applications often need numbers of different types or distributions. The library handles both these needs by defining different distributions that, when used with an engine, produce the desired results. Table 17.16 (overleaf) lists the operations supported by the distribution types.

### Generating Random Real Numbers

Programs often need a source of random floating-point values. In particular, programs frequently need random numbers between zero and one.

The most common, *but incorrect*, way to obtain a random floating-point from `rand` is to divide the result of `rand()` by `RAND_MAX`, which is a system-defined upper limit that is the largest random number that `rand` can return. This technique is incorrect because random integers usually have less precision than floating-point numbers, in which case there are some floating-point values that will never be produced as output.

With the new library facilities, we can easily obtain a floating-point random number. We define an object of type `uniform_real_distribution` and let the library handle mapping random integers to random floating-point numbers. As we did for `uniform_int_distribution`, we specify the minimum and maximum values when we define the object:

```
default_random_engine e; // generates unsigned random integers
// uniformly distributed from 0 to 1 inclusive
uniform_real_distribution<double> u(0,1);
for (size_t i = 0; i < 10; ++i)
    cout << u(e) << " ";
```

This code is nearly identical to the previous program that generated unsigned values. However, because we used a different distribution type, this version generates different results:

```
0.131538 0.45865 0.218959 0.678865 0.934693 0.519416 ...
```

**Table 17.16: Distribution Operations**

|                        |                                                                                                                                                                                                              |
|------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>Dist d;</code>   | Default constructor; makes <code>d</code> ready to use.<br>Other constructors depend on the type of <code>Dist</code> ; see § A.3 (p. 882).<br>The distribution constructors are explicit (§ 7.5.4, p. 296). |
| <code>d(e)</code>      | Successive calls with the same <code>e</code> produce a sequence of random numbers according to the distribution type of <code>d</code> ; <code>e</code> is a random-number engine object.                   |
| <code>d.min()</code>   | Return the smallest and largest numbers <code>d(e)</code> will generate.                                                                                                                                     |
| <code>d.max()</code>   |                                                                                                                                                                                                              |
| <code>d.reset()</code> | Reestablish the state of <code>d</code> so that subsequent uses of <code>d</code> don't depend on values <code>d</code> has already generated.                                                               |

## Using the Distribution's Default Result Type

With one exception, which we'll cover in § 17.4.2 (p. 752), the distribution types are templates that have a single template type parameter that represents the type of the numbers that the distribution generates. These types always generate either a floating-point type or an integral type.

Each distribution template has a default template argument (§ 16.1.3, p. 670). The distribution types that generate floating-point values generate `double` by default. Distributions that generate integral results use `int` as their default. Because the distribution types have only one template parameter, when we want to use the default we must remember to follow the template's name with empty angle brackets to signify that we want the default (§ 16.1.3, p. 671):

```
// empty <> signify we want to use the default result type
uniform_real_distribution<> u(0,1); // generates double by default
```

## Generating Numbers That Are Not Uniformly Distributed

In addition to correctly generating numbers in a specified range, another advantage of the new library is that we can obtain numbers that are nonuniformly distributed. Indeed, the library defines 20 distribution types! These types are listed in § A.3 (p. 882).

As an example, we'll generate a series of normally distributed values and plot the resulting distribution. Because `normal_distribution` generates floating-point numbers, our program will use the `lround` function from the `cmath` header to round each result to its nearest integer. We'll generate 200 numbers centered around a mean of 4 with a standard deviation of 1.5. Because we're using a normal distribution, we can expect all but about 1 percent of the generated numbers to be in the range from 0 to 8, inclusive. Our program will count how many values appear that map to the integers in this range:

```
default_random_engine e;           // generates random integers
normal_distribution<> n(4,1.5); // mean 4, standard deviation 1.5
vector<unsigned> vals(9);        // nine elements each 0
for (size_t i = 0; i != 200; ++i) {
    unsigned v = lround(n(e));   // round to the nearest integer
    if (v < vals.size())         // if this result is in range
        ++vals[v];              // count how often each number appears
}
for (size_t j = 0; j != vals.size(); ++j)
    cout << j << ":" << string(vals[j], '*') << endl;
```

We start by defining our random generator objects and a vector named `vals`. We'll use `vals` to count how often each number in the range 0 ... 9 occurs. Unlike most of our programs that use `vector`, we allocate `vals` at its desired size. By doing so, we start out with each element initialized to 0.

Inside the `for` loop, we call `lround(n(e))` to round the value returned by `n(e)` to the nearest integer. Having obtained the integer that corresponds to our floating-point random number, we use that number to index our vector of counters. Because `n(e)` can produce a number outside the range 0 to 9, we check that the number we got is in range before using it to index `vals`. If the number is in range, we increment the associated counter.

When the loop completes, we print the contents of `vals`, which will generate output such as

```
0: ***
1: *****
2: *********
3: *********
4: *********
5: *********
6: *********
7: *****
8: *
```

Here we print a `string` with as many asterisks as the count of the times the current value was returned by our random-number generator. Note that this figure

is not perfectly symmetrical. If it were, that symmetry should give us reason to suspect the quality of our random-number generator.

### The `bernoulli_distribution` Class

We noted that there was one distribution that does not take a template parameter. That distribution is the `bernoulli_distribution`, which is an ordinary class, not a template. This distribution always returns a `bool` value. It returns `true` with a given probability. By default that probability is .5.

As an example of this kind of distribution, we might have a program that plays a game with a user. To play the game, one of the players—either the user or the program—has to go first. We could use a `uniform_int_distribution` object with a range of 0 to 1 to select the first player. Alternatively, we can use a Bernoulli distribution to make this choice. Assuming that we have a function named `play` that plays the game, we might have a loop such as the following to interact with the user:

```
string resp;
default_random_engine e; // e has state, so it must be outside the loop!
bernoulli_distribution b; // 50/50 odds by default
do {
    bool first = b(e); // if true, the program will go first
    cout << (first ? "We go first"
              : "You get to go first") << endl;
    // play the game passing the indicator of who goes first
    cout << ((play(first)) ? "sorry, you lost"
              : "congrats, you won") << endl;
    cout << "play again? Enter 'yes' or 'no'" << endl;
} while (cin >> resp && resp[0] == 'y');
```

We use a `do while` (§ 5.4.4, p. 189) to repeatedly prompt the user to play.



Because engines return the same sequence of numbers (§ 17.4.1, p. 747), it is essential that we declare engines outside of loops. Otherwise, we'd create a new engine on each iteration and generate the same values on each iteration. Similarly, distributions may retain state and should also be defined outside loops.

One reason to use a `bernoulli_distribution` in this program is that doing so lets us give the program a better chance of going first:

```
bernoulli_distribution b(.55); // give the house a slight edge
```

If we use this definition for `b`, then the program has 55/45 odds of going first.

## 17.5 The IO Library Revisited

In Chapter 8 we introduced the basic architecture and most commonly used parts of the IO library. In this section we'll look at three of the more specialized features that the IO library supports: format control, unformatted IO, and random access.

**EXERCISES SECTION 17.4.2**

**Exercise 17.31:** What would happen if we defined `b` and `e` inside the `do` loop of the game-playing program from this section?

**Exercise 17.32:** What would happen if we defined `resp` inside the loop?

**Exercise 17.33:** Write a version of the word transformation program from § 11.3.6 (p. 440) that allows multiple transformations for a given word and randomly selects which transformation to apply.

### 17.5.1 Formatted Input and Output

In addition to its condition state (§ 8.1.2, p. 312), each `iostream` object also maintains a format state that controls the details of how IO is formatted. The format state controls aspects of formatting such as the notational base for integral values, the precision of floating-point values, the width of an output element, and so on.

The library defines a set of **manipulators** (§ 1.2, p. 7), listed in Tables 17.17 (p. 757) and 17.18 (p. 760), that modify the format state of a stream. A manipulator is a function or object that affects the state of a stream and can be used as an operand to an input or output operator. Like the input and output operators, a manipulator returns the stream object to which it is applied, so we can combine manipulators and data in a single statement.

Our programs have already used one manipulator, `endl`, which we “write” to an output stream as if it were a value. But `endl` isn’t an ordinary value; instead, it performs an operation: It writes a newline and flushes the buffer.

#### Many Manipulators Change the Format State

Manipulators are used for two broad categories of output control: controlling the presentation of numeric values and controlling the amount and placement of padding. Most of the manipulators that change the format state provide set/unset pairs; one manipulator sets the format state to a new value and the other unsets it, restoring the normal default formatting.



Manipulators that change the format state of the stream usually leave the format state changed for all subsequent IO.

The fact that a manipulator makes a persistent change to the format state can be useful when we have a set of IO operations that want to use the same formatting. Indeed, some programs take advantage of this aspect of manipulators to reset the behavior of one or more formatting rules for all its input or output. In such cases, the fact that a manipulator changes the stream is a desirable property.

However, many programs (and, more importantly, programmers) expect the state of the stream to match the normal library defaults. In these cases, leaving the state of the stream in a nonstandard state can lead to errors. As a result, it is usually best to undo whatever state changes are made as soon as those changes are no longer needed.

## Controlling the Format of Boolean Values

One example of a manipulator that changes the formatting state of its object is the `boolalpha` manipulator. By default, `bool` values print as 1 or 0. A `true` value is written as the integer 1 and a `false` value as 0. We can override this formatting by applying the `boolalpha` manipulator to the stream:

```
cout << "default bool values: " << true << " " << false
<< "\nalpha bool values: " << boolalpha
<< true << " " << false << endl;
```

When executed, this program generates the following:

```
default bool values: 1 0
alpha bool values: true false
```

Once we “write” `boolalpha` on `cout`, we’ve changed how `cout` will print `bool` values from this point on. Subsequent operations that print `bools` will print them as either `true` or `false`.

To undo the format state change to `cout`, we apply `noboolalpha`:

```
bool bool_val = get_status();
cout << boolalpha      // sets the internal state of cout
     << bool_val
     << noboolalpha; // resets the internal state to default formatting
```

Here we change the format of `bool` values only to print the value of `bool_val`. Once that value is printed, we immediately reset the stream back to its initial state.

## Specifying the Base for Integral Values

By default, integral values are written and read in decimal notation. We can change the notational base to octal or hexadecimal or back to decimal by using the manipulators `hex`, `oct`, and `dec`:

```
cout << "default: " << 20 << " " << 1024 << endl;
cout << "octal: " << oct << 20 << " " << 1024 << endl;
cout << "hex: " << hex << 20 << " " << 1024 << endl;
cout << "decimal: " << dec << 20 << " " << 1024 << endl;
```

When compiled and executed, this program generates the following output:

```
default: 20 1024
octal: 24 2000
hex: 14 400
decimal: 20 1024
```

Notice that like `boolalpha`, these manipulators change the format state. They affect the immediately following output and all subsequent integral output until the format is reset by invoking another manipulator.



The `hex`, `oct`, and `dec` manipulators affect only integral operands; the representation of floating-point values is unaffected.

## Indicating Base on the Output

By default, when we print numbers, there is no visual cue as to what notational base was used. Is 20, for example, really 20, or an octal representation of 16? When we print numbers in decimal mode, the number is printed as we expect. If we need to print octal or hexadecimal values, it is likely that we should also use the `showbase` manipulator. The `showbase` manipulator causes the output stream to use the same conventions as used for specifying the base of an integral constant:

- A leading `0x` indicates hexadecimal.
- A leading `0` indicates octal.
- The absence of either indicates decimal.

Here we've revised the previous program to use `showbase`:

```
cout << showbase;      // show the base when printing integral values
cout << "default: " << 20 << " " << 1024 << endl;
cout << "in octal: " << oct << 20 << " " << 1024 << endl;
cout << "in hex: " << hex << 20 << " " << 1024 << endl;
cout << "in decimal: " << dec << 20 << " " << 1024 << endl;
cout << noshowbase; // reset the state of the stream
```

The revised output makes it clear what the underlying value really is:

```
default: 20 1024
in octal: 024 02000
in hex: 0x14 0x400
in decimal: 20 1024
```

The `noshowbase` manipulator resets `cout` so that it no longer displays the notational base of integral values.

By default, hexadecimal values are printed in lowercase with a lowercase `x`. We can display the `X` and the hex digits `a-f` as uppercase by applying the `uppercase` manipulator:

```
cout << uppercase << showbase << hex
    << "printed in hexadecimal: " << 20 << " " << 1024
    << nouppercase << noshowbase << dec << endl;
```

This statement generates the following output:

```
printed in hexadecimal: 0X14 0X400
```

We apply the `nouppercase`, `noshowbase`, and `dec` manipulators to return the stream to its original state.

## Controlling the Format of Floating-Point Values

We can control three aspects of floating-point output:

- How many digits of precision are printed

- Whether the number is printed in hexadecimal, fixed decimal, or scientific notation
- Whether a decimal point is printed for floating-point values that are whole numbers

By default, floating-point values are printed using six digits of precision; the decimal point is omitted if the value has no fractional part; and they are printed in either fixed decimal or scientific notation depending on the value of the number. The library chooses a format that enhances readability of the number. Very large and very small values are printed using scientific notation. Other values are printed in fixed decimal.

## Specifying How Much Precision to Print

By default, precision controls the total number of digits that are printed. When printed, floating-point values are rounded, not truncated, to the current precision. Thus, if the current precision is four, then 3.14159 becomes 3.142; if the precision is three, then it is printed as 3.14.

We can change the precision by calling the `precision` member of an IO object or by using the `setprecision` manipulator. The `precision` member is overloaded (§ 6.4, p. 230). One version takes an `int` value and sets the precision to that new value. It returns the *previous* precision value. The other version takes no arguments and returns the current precision value. The `setprecision` manipulator takes an argument, which it uses to set the precision.



The `setprecision` manipulators and other manipulators that take arguments are defined in the `iomanip` header.

The following program illustrates the different ways we can control the precision used to print floating-point values:

```
// cout.precision reports the current precision value
cout << "Precision: " << cout.precision()
       << ", Value: " << sqrt(2.0) << endl;

// cout.precision(12) asks that 12 digits of precision be printed
cout.precision(12);
cout << "Precision: " << cout.precision()
       << ", Value: " << sqrt(2.0) << endl;

// alternative way to set precision using the setprecision manipulator
cout << setprecision(3);
cout << "Precision: " << cout.precision()
       << ", Value: " << sqrt(2.0) << endl;
```

When compiled and executed, the program generates the following output:

```
Precision: 6, Value: 1.41421
Precision: 12, Value: 1.41421356237
Precision: 3, Value: 1.41
```

**Table 17.17: Manipulators Defined in `iostream`**

|                            |                                                                 |
|----------------------------|-----------------------------------------------------------------|
| <code>boolalpha</code>     | Display true and false as strings                               |
| * <code>noboolalpha</code> | Display true and false as 0, 1                                  |
| <code>showbase</code>      | Generate prefix indicating the numeric base of integral values  |
| * <code>noshowbase</code>  | Do not generate notational base prefix                          |
| <code>showpoint</code>     | Always display a decimal point for floating-point values        |
| * <code>noshowpoint</code> | Display a decimal point only if the value has a fractional part |
| <code>showpos</code>       | Display + in nonnegative numbers                                |
| * <code>noshowpos</code>   | Do not display + in nonnegative numbers                         |
| <code>uppercase</code>     | Print 0X in hexadecimal, E in scientific                        |
| * <code>nouppercase</code> | Print 0x in hexadecimal, e in scientific                        |
| * <code>dec</code>         | Display integral values in decimal numeric base                 |
| <code>hex</code>           | Display integral values in hexadecimal numeric base             |
| <code>oct</code>           | Display integral values in octal numeric base                   |
| <code>left</code>          | Add fill characters to the right of the value                   |
| <code>right</code>         | Add fill characters to the left of the value                    |
| <code>internal</code>      | Add fill characters between the sign and the value              |
| <code>fixed</code>         | Display floating-point values in decimal notation               |
| <code>scientific</code>    | Display floating-point values in scientific notation            |
| <code>hexfloat</code>      | Display floating-point values in hex (new to C++ 11)            |
| <code>defaultfloat</code>  | Reset the floating-point format to decimal (new to C++ 11)      |
| <code>unitbuf</code>       | Flush buffers after every output operation                      |
| * <code>nounitbuf</code>   | Restore normal buffer flushing                                  |
| * <code>skipws</code>      | Skip whitespace with input operators                            |
| <code>noskipws</code>      | Do not skip whitespace with input operators                     |
| <code>flush</code>         | Flush the <code>ostream</code> buffer                           |
| <code>ends</code>          | Insert null, then flush the <code>ostream</code> buffer         |
| <code>endl</code>          | Insert newline, then flush the <code>ostream</code> buffer      |

\* indicates the default stream state

This program calls the library `sqrt` function, which is found in the `cmath` header. The `sqrt` function is overloaded and can be called on either a `float`, `double`, or `long double` argument. It returns the square root of its argument.

## Specifying the Notation of Floating-Point Numbers



Unless you need to control the presentation of a floating-point number (e.g., to print data in columns or to print data that represents money or a percentage), it is usually best to let the library choose the notation.

We can force a stream to use scientific, fixed, or hexadecimal notation by using the appropriate manipulator. The `scientific` manipulator changes the stream to use scientific notation. The `fixed` manipulator changes the stream to use fixed decimal.

Under the new library, we can also force floating-point values to use hexadecimal format by using `hexfloat`. The new library provides another manipulator, named `defaultfloat`. This manipulator returns the stream to its default state in

C++  
11

which it chooses a notation based on the value being printed.

These manipulators also change the default meaning of the precision for the stream. After executing `scientific`, `fixed`, or `hexfloat`, the precision value controls the number of digits after the decimal point. By default, precision specifies the total number of digits—both before and after the decimal point. Using `fixed` or `scientific` lets us print numbers lined up in columns, with the decimal point in a fixed position relative to the fractional part being printed:

```
cout << "default format: " << 100 * sqrt(2.0) << '\n'
<< "scientific: " << scientific << 100 * sqrt(2.0) << '\n'
<< "fixed decimal: " << fixed << 100 * sqrt(2.0) << '\n'
<< "hexadecimal: " << hexfloat << 100 * sqrt(2.0) << '\n'
<< "use defaults: " << defaultfloat << 100 * sqrt(2.0)
<< "\n\n";
```

produces the following output:

```
default format: 141.421
scientific: 1.414214e+002
fixed decimal: 141.421356
hexadecimal: 0x1.1ad7bcf+7
use defaults: 141.421
```

By default, the hexadecimal digits and the `e` used in scientific notation are printed in lowercase. We can use the uppercase manipulator to show those values in uppercase.

## Printing the Decimal Point

By default, when the fractional part of a floating-point value is 0, the decimal point is not displayed. The `showpoint` manipulator forces the decimal point to be printed:

```
cout << 10.0 << endl;           // prints 10
cout << showpoint << 10.0      // prints 10.0000
<< noshowpoint << endl;       // revert to default format for the decimal point
```

The `noshowpoint` manipulator reinstates the default behavior. The next output expression will have the default behavior, which is to suppress the decimal point if the floating-point value has a 0 fractional part.

## Padding the Output

When we print data in columns, we often need fairly fine control over how the data are formatted. The library provides several manipulators to help us accomplish the control we might need:

- `setw` to specify the minimum space for the *next* numeric or string value.
- `left` to left-justify the output.
- `right` to right-justify the output. Output is right-justified by default.

- `internal` controls placement of the sign on negative values. `internal` left-justifies the sign and right-justifies the value, padding any intervening space with blanks.
- `setfill` lets us specify an alternative character to use to pad the output. By default, the value is a space.



`setw`, like `endl`, does not change the internal state of the output stream. It determines the size of only the *next* output.

The following program illustrates these manipulators:

```
int i = -16;
double d = 3.14159;

// pad the first column to use a minimum of 12 positions in the output
cout << "i: " << setw(12) << i << "next col" << '\n'
    << "d: " << setw(12) << d << "next col" << '\n';

// pad the first column and left-justify all columns
cout << left
    << "i: " << setw(12) << i << "next col" << '\n'
    << "d: " << setw(12) << d << "next col" << '\n'
    << right;           // restore normal justification

// pad the first column and right-justify all columns
cout << right
    << "i: " << setw(12) << i << "next col" << '\n'
    << "d: " << setw(12) << d << "next col" << '\n';

// pad the first column but put the padding internal to the field
cout << internal
    << "i: " << setw(12) << i << "next col" << '\n'
    << "d: " << setw(12) << d << "next col" << '\n';

// pad the first column, using # as the pad character
cout << setfill('#')
    << "i: " << setw(12) << i << "next col" << '\n'
    << "d: " << setw(12) << d << "next col" << '\n'
    << setfill(' '); // restore the normal pad character
```

When executed, this program generates

```
i:      -16next col
d:      3.14159next col
i: -16      next col
d: 3.14159      next col
i:      -16next col
d:      3.14159next col
i: -      16next col
d:      3.14159next col
i: -#####16next col
d: #####3.14159next col
```

**Table 17.18: Manipulators Defined in `iomanip`**

|                              |                                     |
|------------------------------|-------------------------------------|
| <code>setfill(ch)</code>     | Fill whitespace with ch             |
| <code>setprecision(n)</code> | Set floating-point precision to n   |
| <code>setw(w)</code>         | Read or write value to w characters |
| <code>setbase(b)</code>      | Output integers in base b           |

## Controlling Input Formatting

By default, the input operators ignore whitespace (blank, tab, newline, formfeed, and carriage return). The following loop

```
char ch;
while (cin >> ch)
    cout << ch;
```

given the input sequence

```
a b      c
d
```

executes four times to read the characters a through d, skipping the intervening blanks, possible tabs, and newline characters. The output from this program is

```
abcd
```

The `noskipws` manipulator causes the input operator to read, rather than skip, whitespace. To return to the default behavior, we apply the `skipws` manipulator:

```
cin >> noskipws; // set cin so that it reads whitespace
while (cin >> ch)
    cout << ch;
cin >> skipws; // reset cin to the default state so that it discards whitespace
```

Given the same input as before, this loop makes seven iterations, reading whitespace as well as the characters in the input. This loop generates

```
a b      c
d
```

### EXERCISES SECTION 17.5.1

**Exercise 17.34:** Write a program that illustrates the use of each manipulator in Tables 17.17 (p. 757) and 17.18.

**Exercise 17.35:** Write a version of the program from page 758, that printed the square root of 2 but this time print hexadecimal digits in uppercase.

**Exercise 17.36:** Modify the program from the previous exercise to print the various floating-point values so that they line up in a column.

## 17.5.2 Unformatted Input/Output Operations

So far, our programs have used only **formatted IO** operations. The input and output operators (`<<` and `>>`) format the data they read or write according to the type being handled. The input operators ignore whitespace; the output operators apply padding, precision, and so on.

The library also provides a set of low-level operations that support **unformatted IO**. These operations let us deal with a stream as a sequence of uninterpreted bytes.

### Single-Byte Operations

Several of the unformatted operations deal with a stream one byte at a time. These operations, which are described in Table 17.19, read rather than ignore whitespace. For example, we can use the unformatted IO operations `get` and `put` to read and write the characters one at a time:

```
char ch;
while (cin.get(ch))
    cout.put(ch);
```

This program preserves the whitespace in the input. Its output is identical to the input. It executes the same way as the previous program that used `noskipws`.

Table 17.19: Single-Byte Low-Level IO Operations

|                             |                                                                                                                          |
|-----------------------------|--------------------------------------------------------------------------------------------------------------------------|
| <code>is.get(ch)</code>     | Put the next byte from the <code>istream</code> <code>is</code> in character <code>ch</code> . Returns <code>is</code> . |
| <code>os.put(ch)</code>     | Put the character <code>ch</code> onto the <code>ostream</code> <code>os</code> . Returns <code>os</code> .              |
| <code>is.get()</code>       | Returns next byte from <code>is</code> as an <code>int</code> .                                                          |
| <code>is.putback(ch)</code> | Put the character <code>ch</code> back on <code>is</code> ; returns <code>is</code> .                                    |
| <code>is.unget()</code>     | Move <code>is</code> back one byte; returns <code>is</code> .                                                            |
| <code>is.peek()</code>      | Return the next byte as an <code>int</code> but doesn't remove it.                                                       |

### Putting Back onto an Input Stream

Sometimes we need to read a character in order to know that we aren't ready for it. In such cases, we'd like to put the character back onto the stream. The library gives us three ways to do so, each of which has subtle differences from the others:

- `peek` returns a copy of the next character on the input stream but does not change the stream. The value returned by `peek` stays on the stream.
- `unget` backs up the input stream so that whatever value was last returned is still on the stream. We can call `unget` even if we do not know what value was last taken from the stream.
- `putback` is a more specialized version of `unget`: It returns the last value read from the stream but takes an argument that must be the same as the one that was last read.

In general, we are guaranteed to be able to put back at most one value before the next read. That is, we are not guaranteed to be able to call `putback` or `unget` successively without an intervening read operation.

## **int Return Values from Input Operations**

The `peek` function and the version of `get` that takes no argument return a character from the input stream as an `int`. This fact can be surprising; it might seem more natural to have these functions return a `char`.

The reason that these functions return an `int` is to allow them to return an end-of-file marker. A given character set is allowed to use every value in the `char` range to represent an actual character. Thus, there is no extra value in that range to use to represent end-of-file.

The functions that return `int` convert the character they return to `unsigned char` and then promote that value to `int`. As a result, even if the character set has characters that map to negative values, the `int` returned from these operations will be a positive value (§ 2.1.2, p. 35). The library uses a negative value to represent end-of-file, which is thus guaranteed to be distinct from any legitimate character value. Rather than requiring us to know the actual value returned, the `cstdio` header defines a `const` named `EOF` that we can use to test if the value returned from `get` is end-of-file. It is essential that we use an `int` to hold the return from these functions:

```
int ch;      // use an int, not a char to hold the return from get()
// loop to read and write all the data in the input
while ((ch = cin.get()) != EOF)
    cout.put(ch);
```

This program operates identically to the one on page 761, the only difference being the version of `get` that is used to read the input.

## **Multi-Byte Operations**

Some unformatted IO operations deal with chunks of data at a time. These operations can be important if speed is an issue, but like other low-level operations, they are error-prone. In particular, these operations require us to allocate and manage the character arrays (§ 12.2, p. 476) used to store and retrieve data. The multi-byte operations are listed in Table 17.20.

The `get` and `getline` functions take the same parameters, and their actions are similar but not identical. In each case, `sink` is a `char` array into which the data are placed. The functions read until one of the following conditions occurs:

- `size - 1` characters are read
- End-of-file is encountered
- The delimiter character is encountered

The difference between these functions is the treatment of the delimiter: `get` leaves the delimiter as the next character of the `istream`, whereas `getline` reads and discards the delimiter. In either case, the delimiter is *not* stored in `sink`.

**Table 17.20: Multi-Byte Low-Level IO Operations**

|                                            |                                                                                                                                                                                                                                                                                                                                                                                                     |
|--------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>is.get(sink, size, delim)</code>     | Reads up to <code>size</code> bytes from <code>is</code> and stores them in the character array beginning at the address pointed to by <code>sink</code> . Reads until encountering the <code>delim</code> character or until it has read <code>size</code> bytes or encounters end-of-file. If <code>delim</code> is present, it is left on the input stream and not read into <code>sink</code> . |
| <code>is.getline(sink, size, delim)</code> | Same behavior as the three-argument version of <code>get</code> but reads and discards <code>delim</code> .                                                                                                                                                                                                                                                                                         |
| <code>is.read(sink, size)</code>           | Reads up to <code>size</code> bytes into the character array <code>sink</code> . Returns <code>is</code> .                                                                                                                                                                                                                                                                                          |
| <code>is.gcount()</code>                   | Returns number of bytes read from the stream <code>is</code> by the last call to an unformatted read operation.                                                                                                                                                                                                                                                                                     |
| <code>os.write(source, size)</code>        | Writes <code>size</code> bytes from the character array <code>source</code> to <code>os</code> . Returns <code>os</code> .                                                                                                                                                                                                                                                                          |
| <code>is.ignore(size, delim)</code>        | Reads and ignores at most <code>size</code> characters up to and including <code>delim</code> . Unlike the other unformatted functions, <code>ignore</code> has default arguments: <code>size</code> defaults to 1 and <code>delim</code> to end-of-file.                                                                                                                                           |



It is a common error to intend to remove the delimiter from the stream but to forget to do so.

### Determining How Many Characters Were Read

Several of the read operations read an unknown number of bytes from the input. We can call `gcount` to determine how many characters the last unformatted input operation read. It is essential to call `gcount` before any intervening unformatted input operation. In particular, the single-character operations that put characters back on the stream are also unformatted input operations. If `peek`, `unget`, or `putback` are called before calling `gcount`, then the return value will be 0.

#### 17.5.3 Random Access to a Stream

The various stream types generally support random access to the data in their associated stream. We can reposition the stream so that it skips around, reading first the last line, then the first, and so on. The library provides a pair of functions to *seek* to a given location and to *tell* the current location in the associated stream.



Random IO is an inherently system-dependent. To understand how to use these features, you must consult your system's documentation.

Although these `seek` and `tell` functions are defined for all the stream types, whether they do anything useful depends on the device to which the stream is bound. On most systems, the streams bound to `cin`, `cout`, `cerr`, and `clog` do

### **CAUTION: LOW-LEVEL ROUTINES ARE ERROR-PRONE**

In general, we advocate using the higher-level abstractions provided by the library. The IO operations that return `int` are a good example of why.

It is a common programming error to assign the return, from `get` or `peek` to a `char` rather than an `int`. Doing so is an error, but an error the compiler will not detect. Instead, what happens depends on the machine and on the input data. For example, on a machine in which `chars` are implemented as `unsigned chars`, this loop will run forever:

```
char ch;      // using a char here invites disaster!
// the return from cin.get is converted to char and then compared to an int
while ((ch = cin.get()) != EOF)
    cout.put(ch);
```

The problem is that when `get` returns `EOF`, that value will be converted to an `unsigned char` value. That converted value is no longer equal to the `int` value of `EOF`, and the loop will continue forever. Such errors are likely to be caught in testing.

On machines for which `chars` are implemented as `signed chars`, we can't say with confidence what the behavior of the loop might be. What happens when an out-of-bounds value is assigned to a `signed` value is up to the compiler. On many machines, this loop will appear to work, unless a character in the input matches the `EOF` value. Although such characters are unlikely in ordinary data, presumably low-level IO is necessary only when we read binary values that do not map directly to ordinary characters and numeric values. For example, on our machine, if the input contains a character whose value is '`\377`', then the loop terminates prematurely. '`\377`' is the value on our machine to which `-1` converts when used as a `signed char`. If the input has this value, then it will be treated as the (premature) end-of-file indicator.

Such bugs do not happen when we read and write typed values. If you can use the more type-safe, higher-level operations supported by the library, do so.

### **EXERCISES SECTION 17.5.2**

**Exercise 17.37:** Use the unformatted version of `getline` to read a file a line at a time. Test your program by giving it a file that contains empty lines as well as lines that are longer than the character array that you pass to `getline`.

**Exercise 17.38:** Extend your program from the previous exercise to print each word you read onto its own line.

*not* support random access—after all, what would it mean to jump back ten places when we're writing directly to `cout`? We can call the `seek` and `tell` functions, but these functions will fail at run time, leaving the stream in an invalid state.



Because the `istream` and `ostream` types usually do not support random access, the remainder of this section should be considered as applicable to only the `fstream` and `sstream` types.

## Seek and Tell Functions

To support random access, the IO types maintain a marker that determines where the next read or write will happen. They also provide two functions: One repositions the marker by *seeking* to a given position; the second *tells* us the current position of the marker. The library actually defines two pairs of *seek* and *tell* functions, which are described in Table 17.21. One pair is used by input streams, the other by output streams. The input and output versions are distinguished by a suffix that is either a *g* or a *p*. The *g* versions indicate that we are “getting” (reading) data, and the *p* functions indicate that we are “putting” (writing) data.

**Table 17.21: Seek and Tell Functions**

|                               |                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|-------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>tellg()</code>          | Return the current position of the marker in an input stream ( <code>tellg</code> ) or an output stream ( <code>tellp</code> ).                                                                                                                                                                                                                                                                                                         |
| <code>seekg(pos)</code>       | Reposition the marker in an input or output stream to the given absolute address in the stream. <code>pos</code> is usually a value returned by a previous call to the corresponding <code>tellg</code> or <code>tellp</code> function.                                                                                                                                                                                                 |
| <code>seekp(pos)</code>       | Reposition the marker for an input or output stream integral number <code>off</code> characters ahead or behind <code>from</code> . <code>from</code> can be one of <ul style="list-style-type: none"><li>• <code>beg</code>, seek relative to the beginning of the stream</li><li>• <code>cur</code>, seek relative to the current position of the stream</li><li>• <code>end</code>, seek relative to the end of the stream</li></ul> |
| <code>seekg(off, from)</code> | Reposition the marker for an input or output stream integral number <code>off</code> characters ahead or behind <code>from</code> . <code>from</code> can be one of <ul style="list-style-type: none"><li>• <code>beg</code>, seek relative to the beginning of the stream</li><li>• <code>cur</code>, seek relative to the current position of the stream</li><li>• <code>end</code>, seek relative to the end of the stream</li></ul> |
| <code>seekp(off, from)</code> | Reposition the marker for an input or output stream integral number <code>off</code> characters ahead or behind <code>from</code> . <code>from</code> can be one of <ul style="list-style-type: none"><li>• <code>beg</code>, seek relative to the beginning of the stream</li><li>• <code>cur</code>, seek relative to the current position of the stream</li><li>• <code>end</code>, seek relative to the end of the stream</li></ul> |

Logically enough, we can use only the *g* versions on an *istream* and on the types *ifstream* and *istringstream* that inherit from *istream* (§ 8.1, p. 311). We can use only the *p* versions on an *ostream* and on the types that inherit from it, *ofstream* and *ostringstream*. An *iostream*, *fstream*, or *stringstream* can both read and write the associated stream; we can use either the *g* or *p* versions on objects of these types.

## There Is Only One Marker

The fact that the library distinguishes between the “putting” and “getting” versions of the seek and tell functions can be misleading. Even though the library makes this distinction, it maintains only a single marker in a stream—there is *not* a distinct read marker and write marker.

When we’re dealing with an input-only or output-only stream, the distinction isn’t even apparent. We can use only the *g* or only the *p* versions on such streams. If we attempt to call `tellp` on an *ifstream*, the compiler will complain. Similarly, it will not let us call `seekg` on an *ostringstream*.

The *fstream* and *stringstream* types can read and write the same stream. In these types there is a single buffer that holds data to be read and written and a single marker denoting the current position in the buffer. The library maps both the *g* and *p* positions to this single marker.



Because there is only a single marker, we *must* do a seek to reposition the marker whenever we switch between reading and writing.

## Repositioning the Marker

There are two versions of the seek functions: One moves to an “absolute” address within the file; the other moves to a byte offset from a given position:

```
// set the marker to a fixed position
seekg(new_position);    // set the read marker to the given pos_type location
seekp(new_position);    // set the write marker to the given pos_type location

// offset some distance ahead of or behind the given starting point
seekg(offset, from);    // set the read marker offset distance from from
seekp(offset, from);    // offset has type off_type
```

The possible values for `from` are listed in Table 17.21 (on the previous page).

The arguments, `new_position` and `offset`, have machine-dependent types named `pos_type` and `off_type`, respectively. These types are defined in both `istream` and `ostream`. `pos_type` represents a file position and `off_type` represents an offset from that position. A value of type `off_type` can be positive or negative; we can seek forward or backward in the file.

## Accessing the Marker

The `tellg` or `tellp` functions return a `pos_type` value denoting the current position of the stream. The tell functions are usually used to remember a location so that we can subsequently seek back to it:

```
// remember the current write position in mark
ostringstream writeStr;    // output stringstream
ostringstream::pos_type mark = writeStr.tellp();
// ...
if (cancelEntry)
    // return to the remembered position
    writeStr.seekp(mark);
```

## Reading and Writing to the Same File

Let’s look at a programming example. Assume we are given a file to read. We are to write a newline at the end of the file that contains the relative position at which each line begins. For example, given the following file,

```
abcd
efg
hi
j
```

the program should produce the following modified file:

```
abcd
efg
hi
j
5 9 12 14
```

Note that our program need not write the offset for the first line—it always occurs at position 0. Also note that the offset counts must include the invisible newline character that ends each line. Finally, note that the last number in the output is the offset for the line on which our output begins. By including this offset in our output, we can distinguish our output from the file’s original contents. We can read the last number in the resulting file and seek to the corresponding offset to get to the beginning of our output.

Our program will read the file a line at a time. For each line, we’ll increment a counter, adding the size of the line we just read. That counter is the offset at which the next line starts:

```
int main()
{
    // open for input and output and preposition file pointers to end-of-file
    // file mode argument see § 8.4 (p. 319)
    fstream inOut("copyOut",
                  fstream::ate | fstream::in | fstream::out);
    if (!inOut) {
        cerr << "Unable to open file!" << endl;
        return EXIT_FAILURE; // EXIT_FAILURE see § 6.3.2 (p. 227)
    }
    // inOut is opened in ate mode, so it starts out positioned at the end
    auto end_mark = inOut.tellg(); // remember original end-of-file position
    inOut.seekg(0, fstream::beg); // reposition to the start of the file
    size_t cnt = 0; // accumulator for the byte count
    string line; // hold each line of input
    // while we haven't hit an error and are still reading the original data
    while (inOut && inOut.tellg() != end_mark
          && getline(inOut, line)) { // and can get another line of input
        cnt += line.size() + 1; // add 1 to account for the newline
        auto mark = inOut.tellg(); // remember the read position
        inOut.seekp(0, fstream::end); // set the write marker to the end
        inOut << cnt; // write the accumulated length
        // print a separator if this is not the last line
        if (mark != end_mark) inOut << " ";
        inOut.seekg(mark); // restore the read position
    }
    inOut.seekp(0, fstream::end); // seek to the end
    inOut << "\n"; // write a newline at end-of-file
    return 0;
}
```

Our program opens its `fstream` using the `in`, `out`, and `ate` modes (§ 8.4, p. 319). The first two modes indicate that we intend to read and write the same file. Speci-

**EXERCISES SECTION 18.1.5**

**Exercise 18.9:** Define the bookstore exception classes described in this section and rewrite your `Sales_data` compound assignment operator to throw an exception.

**Exercise 18.10:** Write a program that uses the `Sales_data` addition operator on objects that have differing ISBNs. Write two versions of the program: one that handles the exception and one that does not. Compare the behavior of the programs so that you become familiar with what happens when an uncaught exception occurs.

**Exercise 18.11:** Why is it important that the `what` function doesn't throw?

## 18.2 Namespaces

Large programs tend to use independently developed libraries. Such libraries also tend to define a large number of global names, such as classes, functions, and templates. When an application uses libraries from many different vendors, it is almost inevitable that some of these names will clash. Libraries that put names into the global namespace are said to cause **namespace pollution**.

Traditionally, programmers avoided namespace pollution by using very long names for the global entities they defined. Those names often contained a prefix indicating which library defined the name:

```
class cplusplus_primer_Query { ... };
string cplusplus_primer_make_plural(size_t, string&);
```

This solution is far from ideal: It can be cumbersome for programmers to write and read programs that use such long names.

**Namespaces** provide a much more controlled mechanism for preventing name collisions. Namespaces partition the global namespace. A namespace is a scope. By defining a library's names inside a namespace, library authors (and users) can avoid the limitations inherent in global names.

### 18.2.1 Namespace Definitions

A namespace definition begins with the keyword `namespace` followed by the namespace name. Following the namespace name is a sequence of declarations and definitions delimited by curly braces. Any declaration that can appear at global scope can be put into a namespace: classes, variables (with their initializations), functions (with their definitions), templates, and other namespaces:

```
namespace cplusplus_primer {
    class Sales_data { /* ... */;
    Sales_data operator+(const Sales_data&,
                          const Sales_data&);
    class Query { /* ... */;
    class Query_base { /* ... */;
} // like blocks, namespaces do not end with a semicolon
```

This code defines a namespace named `cplusplus_primer` with four members: three classes and an overloaded `+` operator.

As with any name, a namespace name must be unique within the scope in which the namespace is defined. Namespaces may be defined at global scope or inside another namespace. They may not be defined inside a function or a class.



A namespace scope does not end with a semicolon.

## Each Namespace Is a Scope

As is the case for any scope, each name in a namespace must refer to a unique entity within that namespace. Because different namespaces introduce different scopes, different namespaces may have members with the same name.

Names defined in a namespace may be accessed directly by other members of the namespace, including scopes nested within those members. Code outside the namespace must indicate the namespace in which the name is defined:

```
cplusplus_primer::Query q =
    cplusplus_primer::Query("hello");
```

If another namespace (say, `AddisonWesley`) also provides a `Query` class and we want to use that class instead of the one defined in `cplusplus_primer`, we can do so by modifying our code as follows:

```
AddisonWesley::Query q = AddisonWesley::Query("hello");
```

## Namespaces Can Be Discontiguous

As we saw in § 16.5 (p. 709), unlike other scopes, a namespace can be defined in several parts. Writing a namespace definition:

```
namespace nsp {
// declarations
}
```

either defines a new namespace named `nsp` or adds to an existing one. If the name `nsp` does not refer to a previously defined namespace, then a new namespace with that name is created. Otherwise, this definition opens an existing namespace and adds declarations to that already existing namespace.

The fact that namespace definitions can be discontiguous lets us compose a namespace from separate interface and implementation files. Thus, a namespace can be organized in the same way that we manage our own class and function definitions:

- Namespace members that define classes, and declarations for the functions and objects that are part of the class interface, can be put into header files. These headers can be included by files that use those namespace members.
- The definitions of namespace members can be put in separate source files.

Organizing our namespaces this way also satisfies the requirement that various entities—non-inline functions, static data members, variables, and so forth—may be defined only once in a program. This requirement applies equally to names defined in a namespace. By separating the interface and implementation, we can ensure that the functions and other names we need are defined only once, but the same declaration will be seen whenever the entity is used.



Namespaces that define multiple, unrelated types should use separate files to represent each type (or each collection of related types) that the namespace defines.

## Defining the Primer Namespace

Using this strategy for separating interface and implementation, we might define the `cplusplus_primer` library in several separate files. The declarations for `Sales_data` and its related functions would be placed in `Sales_data.h`, those for the `Query` classes of Chapter 15 in `Query.h`, and so on. The corresponding implementation files would be in files such as `Sales_data.cc` and `Query.cc`:

```
// ---- Sales_data.h ----
// #includes should appear before opening the namespace
#include <string>
namespace cplusplus_primer {
    class Sales_data { /* ... */;
    Sales_data operator+(const Sales_data&,
                          const Sales_data&);
    // declarations for the remaining functions in the Sales_data interface
}
// ---- Sales_data.cc ----
// be sure any #includes appear before opening the namespace
#include "Sales_data.h"
namespace cplusplus_primer {
    // definitions for Sales_data members and overloaded operators
}
```

A program using our library would include whichever headers it needed. The names in those headers are defined inside the `cplusplus_primer` namespace:

```
// ---- user.cc ----
// names in the Sales_data.h header are in the cplusplus_primer namespace
#include "Sales_data.h"

int main()
{
    using cplusplus_primer::Sales_data;
    Sales_data transl, trans2;
    // ...
    return 0;
}
```

This program organization gives the developers and the users of our library the needed modularity. Each class is still organized into its own interface and

implementation files. A user of one class need not compile names related to the others. We can hide the implementations from our users, while allowing the files `Sales_data.cc` and `user.cc` to be compiled and linked into one program without causing any compile-time or link-time errors. Developers of the library can work independently on the implementation of each type.

It is worth noting that ordinarily, we do not put a `#include` inside the namespace. If we did, we would be attempting to define all the names in that header as members of the enclosing namespace. For example, if our `Sales_data.h` file opened the `cplusplus_primer` before including the `string` header our program would be in error. It would be attempting to define the `std` namespace nested inside `cplusplus_primer`.

## Defining Namespace Members

Assuming the appropriate declarations are in scope, code inside a namespace may use the short form for names defined in the same (or in an enclosing) namespace:

```
#include "Sales_data.h"
namespace cplusplus_primer {    // reopen cplusplus_primer
    // members defined inside the namespace may use unqualified names
    std::istream&
    operator>>(std::istream& in, Sales_data& s) { /* ... */ }
}
```

It is also possible to define a namespace member outside its namespace definition. The namespace declaration of the name must be in scope, and the definition must specify the namespace to which the name belongs:

```
// namespace members defined outside the namespace must use qualified names
cplusplus_primer::Sales_data
cplusplus_primer::operator+(const Sales_data& lhs,
                           const Sales_data& rhs)
{
    Sales_data ret(lhs);
    // ...
}
```

As with class members defined outside a class, once the fully qualified name is seen, we are in the scope of the namespace. Inside the `cplusplus_primer` namespace, we can use other namespace member names without qualification. Thus, even though `Sales_data` is a member of the `cplusplus_primer` namespace, we can use its unqualified name to define the parameters in this function.

Although a namespace member can be defined outside its namespace, such definitions must appear in an enclosing namespace. That is, we can define the `Sales_data operator+` inside the `cplusplus_primer` namespace or at global scope. We cannot define this operator in an unrelated namespace.

## Template Specializations

Template specializations must be defined in the same namespace that contains the original template (§ 16.5, p. 709). As with any other namespace name, so long as

we have declared the specialization inside the namespace, we can define it outside the namespace:

```
// we must declare the specialization as a member of std
namespace std {
    template <> struct hash<Sales_data>;
}

// having added the declaration for the specialization to std
// we can define the specialization outside the std namespace
template <> struct std::hash<Sales_data>
{
    size_t operator()(const Sales_data& s) const
    { return hash<string>()(s.bookNo) ^
           hash<unsigned>()(s.units_sold) ^
           hash<double>()(s.revenue); }
    // other members as before
};


```

## The Global Namespace

Names defined at global scope (i.e., names declared outside any class, function, or namespace) are defined inside the **global namespace**. The global namespace is implicitly declared and exists in every program. Each file that defines entities at global scope (implicitly) adds those names to the global namespace.

The scope operator can be used to refer to members of the global namespace. Because the global namespace is implicit, it does not have a name; the notation

```
::member_name
```

refers to a member of the global namespace.

## Nested Namespaces

A nested namespace is a namespace defined inside another namespace:

```
namespace cplusplus_primer {
    // first nested namespace: defines the Query portion of the library
    namespace QueryLib {
        class Query { /* ... */ };
        Query operator&(const Query&, const Query&);
        // ...
    }
    // second nested namespace: defines the Sales_data portion of the library
    namespace Bookstore {
        class Quote { /* ... */ };
        class Disc_quote : public Quote { /* ... */ };
        // ...
    }
}
```

The `cplusplus_primer` namespace now contains two nested namespaces: the namespaces named `QueryLib` and `Bookstore`.

A nested namespace is a nested scope—its scope is nested within the namespace that contains it. Nested namespace names follow the normal rules: Names declared in an inner namespace hide declarations of the same name in an outer namespace. Names defined inside a nested namespace are local to that inner namespace. Code in the outer parts of the enclosing namespace may refer to a name in a nested namespace only through its qualified name: For example, the name of the class declared in the nested namespace `QueryLib` is

```
cplusplus_primer::QueryLib::Query
```

## Inline Namespaces

**C++ 11** The new standard introduced a new kind of nested namespace, an **inline namespace**. Unlike ordinary nested namespaces, names in an inline namespace can be used as if they were direct members of the enclosing namespace. That is, we need not qualify names from an inline namespace by their namespace name. We can access them using only the name of the enclosing namespace.

An inline namespace is defined by preceding the keyword `namespace` with the keyword `inline`:

```
inline namespace FifthEd {
    // namespace for the code from the Primer Fifth Edition
}
namespace FifthEd { // implicitly inline
    class Query_base { /* ... */;
        // other Query-related declarations
}
```

The keyword must appear on the first definition of the namespace. If the namespace is later reopened, the keyword `inline` need not be, but may be, repeated.

Inline namespaces are often used when code changes from one release of an application to the next. For example, we can put all the code from the current edition of the Primer into an inline namespace. Code for previous versions would be in non-inlined namespaces:

```
namespace FourthEd {
    class Item_base { /* ... */;
    class Query_base { /* ... */;
        // other code from the Fourth Edition
}
```

The overall `cplusplus_primer` namespace would include the definitions of both namespaces. For example, assuming that each namespace was defined in a header with the corresponding name, we'd define `cplusplus_primer` as follows:

```
namespace cplusplus_primer {
#include "FifthEd.h"
#include "FourthEd.h"
}
```

Because `FifthEd` is inline, code that refers to `cplusplus_primer::` will get the version from that namespace. If we want the earlier edition code, we can access it as we would any other nested namespace, by using the names of all the enclosing namespaces: for example, `cplusplus_primer::FourthEd::Query_base`.

## Unnamed Namespaces

An **unnamed namespace** is the keyword `namespace` followed immediately by a block of declarations delimited by curly braces. Variables defined in an unnamed namespace have static lifetime: They are created before their first use and destroyed when the program ends.

An unnamed namespace may be discontiguous within a given file but does not span files. Each file has its own unnamed namespace. If two files contain unnamed namespaces, those namespaces are unrelated. Both unnamed namespaces can define the same name; those definitions would refer to different entities. If a header defines an unnamed namespace, the names in that namespace define different entities local to each file that includes the header.



Unlike other namespaces, an unnamed namespace is local to a particular file and never spans multiple files.

Names defined in an unnamed namespace are used directly; after all, there is no namespace name with which to qualify them. It is not possible to use the scope operator to refer to members of unnamed namespaces.

Names defined in an unnamed namespace are in the same scope as the scope at which the namespace is defined. If an unnamed namespace is defined at the outermost scope in the file, then names in the unnamed namespace must differ from names defined at global scope:

```
int i;      // global declaration for i
namespace {
    int i;
}
// ambiguous: defined globally and in an unnested, unnamed namespace
i = 10;
```

In all other ways, the members of an unnamed namespace are normal program entities. An unnamed namespace, like any other namespace, may be nested inside another namespace. If the unnamed namespace is nested, then names in it are accessed in the normal way, using the enclosing namespace name(s):

```
namespace local {
    namespace {
        int i;
    }
}
// ok: i defined in a nested unnamed namespace is distinct from global i
local::i = 42;
```

**UNNAMED NAMESPACES REPLACE FILE STATICS**

Prior to the introduction of namespaces, programs declared names as `static` to make them local to a file. The use of *file statics* is inherited from C. In C, a global entity declared `static` is invisible outside the file in which it is declared.



The use of file `static` declarations is deprecated by the C++ standard. File statics should be avoided and unnamed namespaces used instead.

**EXERCISES SECTION 18.2.1**

**Exercise 18.12:** Organize the programs you have written to answer the questions in each chapter into their own namespaces. That is, namespace `chapter15` would contain code for the `Query` programs and `chapter10` would contain the `TextQuery` code. Using this structure, compile the `Query` code examples.

**Exercise 18.13:** When might you use an unnamed namespace?

**Exercise 18.14:** Suppose we have the following declaration of the `operator*` that is a member of the nested namespace `mathLib::MatrixLib`:

```
namespace mathLib {
    namespace MatrixLib {
        class matrix { /* ... */ };
        matrix operator*
            (const matrix &, const matrix &);
        // ...
    }
}
```

How would you declare this operator in global scope?

**18.2.2 Using Namespace Members**

Referring to namespace members as `namespace_name::member_name` is admittedly cumbersome, especially if the namespace name is long. Fortunately, there are ways to make it easier to use namespace members. Our programs have used one of these ways, using declarations (§ 3.1, p. 82). The others, namespace aliases and using directives, will be described in this section.

**Namespace Aliases**

A **namespace alias** can be used to associate a shorter synonym with a namespace name. For example, a long namespace name such as

```
namespace cplusplus_primer { /* ... */ };
```

can be associated with a shorter synonym as follows:

```
namespace primer = cplusplus_primer;
```

A namespace alias declaration begins with the keyword `namespace`, followed by the alias name, followed by the `=` sign, followed by the original namespace name and a semicolon. It is an error if the original namespace name has not already been defined as a namespace.

A namespace alias can also refer to a nested namespace:

```
namespace Qlib = cplusplus_primer::QueryLib;  
Qlib::Query q;
```



A namespace can have many synonyms, or aliases. All the aliases and the original namespace name can be used interchangeably.

## using Declarations: A Recap

A **using declaration** introduces only one namespace member at a time. It allows us to be very specific regarding which names are used in our programs.

Names introduced in a **using declaration** obey normal scope rules: They are visible from the point of the **using declaration** to the end of the scope in which the declaration appears. Entities with the same name defined in an outer scope are hidden. The unqualified name may be used only within the scope in which it is declared and in scopes nested within that scope. Once the scope ends, the fully qualified name must be used.

A **using declaration** can appear in global, local, namespace, or class scope. In class scope, such declarations may only refer to a base class member (§ 15.5, p. 615).

## using Directives

A **using directive**, like a **using declaration**, allows us to use the unqualified form of a namespace name. Unlike a **using declaration**, we retain no control over which names are made visible—they all are.

A **using directive** begins with the keyword `using`, followed by the keyword `namespace`, followed by a namespace name. It is an error if the name is not a previously defined namespace name. A **using directive** may appear in global, local, or namespace scope. It may not appear in a class scope.

These directives make all the names from a specific namespace visible without qualification. The short form names can be used from the point of the **using directive** to the end of the scope in which the **using directive** appears.



Providing a **using directive** for namespaces, such as `std`, that our application does not control reintroduces all the name collision problems inherent in using multiple libraries.

## using Directives and Scope

The scope of names introduced by a **using directive** is more complicated than the scope of names in **using declarations**. As we've seen, a **using declaration** puts the name in the same scope as that of the **using declaration** itself. It is as if the **using declaration** declares a local alias for the namespace member.

A `using` directive does not declare local aliases. Rather, it has the effect of lifting the namespace members into the nearest scope that contains both the namespace itself and the `using` directive.

This difference in scope between a `using` declaration and a `using` directive stems directly from how these two facilities work. In the case of a `using` declaration, we are simply making name directly accessible in the local scope. In contrast, a `using` directive makes the entire contents of a namespace available. In general, a namespace might include definitions that cannot appear in a local scope. As a consequence, a `using` directive is treated as if it appeared in the nearest enclosing namespace scope.

In the simplest case, assume we have a namespace `A` and a function `f`, both defined at global scope. If `f` has a `using` directive for `A`, then in `f` it will be as if the names in `A` appeared in the global scope prior to the definition of `f`:

```
// namespace A and function f are defined at global scope
namespace A {
    int i, j;
}
void f()
{
    using namespace A;      // injects the names from A into the global scope
    cout << i * j << endl; // uses i and j from namespace A
    // ...
}
```

## using Directives Example

Let's look at an example:

```
namespace blip {
    int i = 16, j = 15, k = 23;
    // other declarations
}
int j = 0; // ok: j inside blip is hidden inside a namespace
void manip()
{
    // using directive; the names in blip are "added" to the global scope
    using namespace blip; // clash between ::j and blip::j
    // detected only if j is used
    ++i;           // sets blip::i to 17
    ++j;           // error ambiguous: global j or blip::j?
    ++::j;         // ok: sets global j to 1
    ++blip::j;     // ok: sets blip::j to 16
    int k = 97;   // local k hides blip::k
    ++k;           // sets local k to 98
}
```

The `using` directive in `manip` makes all the names in `blip` directly accessible; code inside `manip` can refer to the names of these members, using their short form.

The members of `blip` appear as if they were defined in the scope in which both `blip` and `manip` are defined. Assuming `manip` is defined at global scope, then the members of `blip` appear as if they were declared in global scope.

When a namespace is injected into an enclosing scope, it is possible for names in the namespace to conflict with other names defined in that (enclosing) scope. For example, inside `manip`, the `blip` member `j` conflicts with the global object named `j`. Such conflicts are permitted, but to use the name, we must explicitly indicate which version is wanted. Any unqualified use of `j` within `manip` is ambiguous.

To use a name such as `j`, we must use the scope operator to indicate which name is wanted. We would write `::j` to obtain the variable defined in global scope. To use the `j` defined in `blip`, we must use its qualified name, `blip::j`.

Because the names are in different scopes, local declarations within `manip` may hide some of the namespace member names. The local variable `k` hides the namespace member `blip::k`. Referring to `k` within `manip` is not ambiguous; it refers to the local variable `k`.

## Headers and using Declarations or Directives

A header that has a `using` directive or declaration at its top-level scope injects names into every file that includes the header. Ordinarily, headers should define only the names that are part of its interface, not names used in its own implementation. As a result, header files should not contain `using` directives or `using` declarations except inside functions or namespaces (§ 3.1, p. 83).

### CAUTION: AVOID USING DIRECTIVES

`using` directives, which inject all the names from a namespace, are deceptively simple to use: With only a single statement, all the member names of a namespace are suddenly visible. Although this approach may seem simple, it can introduce its own problems. If an application uses many libraries, and if the names within these libraries are made visible with `using` directives, then we are back to square one, and the global namespace pollution problem reappears.

Moreover, it is possible that a working program will fail to compile when a new version of the library is introduced. This problem can arise if a new version introduces a name that conflicts with a name that the application is using.

Another problem is that ambiguity errors caused by `using` directives are detected only at the point of use. This late detection means that conflicts can arise long after introducing a particular library. If the program begins using a new part of the library, previously undetected collisions may arise.

Rather than relying on a `using` directive, it is better to use a `using` declaration for each namespace name used in the program. Doing so reduces the number of names injected into the namespace. Ambiguity errors caused by `using` declarations are detected at the point of declaration, not use, and so are easier to find and fix.



One place where `using` directives are useful is in the implementation files of the namespace itself.

## EXERCISES SECTION 18.2.2

**Exercise 18.15:** Explain the differences between using declarations and directives.

**Exercise 18.16:** Explain the following code assuming using declarations for all the members of namespace `Exercise` are located at the location labeled *position 1*. What if they appear at *position 2* instead? Now answer the same question but replace the using declarations with a using directive for namespace `Exercise`.

```
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;
}
```

**Exercise 18.17:** Write code to test your answers to the previous question.

### 18.2.3 Classes, Namespaces, and Scope

Name lookup for names used inside a namespace follows the normal lookup rules: The search looks outward through the enclosing scopes. An enclosing scope might be one or more nested namespaces, ending in the all-encompassing global namespace. Only names that have been declared before the point of use that are in blocks that are still open are considered:

```
namespace A {
    int i;
    namespace B {
        int i;           // hides A::i within B
        int j;
        int f1()
        {
            int j;     // j is local to f1 and hides A::B::j
            return i; // returns B::i
        }
    } // namespace B is closed and names in it are no longer visible
    int f2()
    {
        return j;      // error: j is not defined
    }
    int j = i;       // initialized from A::i
}
```

When a class is wrapped in a namespace, the normal lookup still happens: When a name is used by a member function, look for that name in the member first, then within the class (including base classes), then look in the enclosing scopes, one or more of which might be a namespace:

```
namespace A {
    int i;
    int k;

    class C1 {
        public:
            C1(): i(0), j(0) { }      // ok: initializes C1::i and C1::j
            int f1() { return k; }    // returns A::k
            int f2() { return h; }    // error: h is not defined
            int f3();
        private:
            int i;                  // hides A::i within C1
            int j;
    };
    int h = i;                  // initialized from A::i
}

// member f3 is defined outside class C1 and outside namespace A
int A::C1::f3() { return h; } // ok: returns A::h
```

With the exception of member function definitions that appear inside the class body (§ 7.4.1, p. 283), scopes are always searched upward; names must be declared before they can be used. Hence, the `return` in `f2` will not compile. It attempts to reference the name `h` from namespace `A`, but `h` has not yet been defined. Had that name been defined in `A` before the definition of `C1`, the use of `h` would be legal. Similarly, the use of `h` inside `f3` is okay, because `f3` is defined after `A::h`.



The order in which scopes are examined to find a name can be inferred from the qualified name of a function. The qualified name indicates, in reverse order, the scopes that are searched.

The qualifiers `A::C1::f3` indicate the reverse order in which the class scopes and namespace scopes are to be searched. The first scope searched is that of the function `f3`. Then the class scope of its enclosing class `C1` is searched. The scope of the namespace `A` is searched last before the scope containing the definition of `f3` is examined.

## Argument-Dependent Lookup and Parameters of Class Type



Consider the following simple program:

```
std::string s;
std::cin >> s;
```

As we know, this call is equivalent to (§ 14.1, p. 553):

```
operator>>(std::cin, s);
```

This `operator>>` function is defined by the `string` library, which in turn is defined in the `std` namespace. Yet we can we call `operator>>` without an `std::` qualifier and without a `using` declaration.

We can directly access the output operator because there is an important exception to the rule that names defined in a namespace are hidden. When we pass an object of a class type to a function, the compiler searches the namespace in which the argument's class is defined *in addition* to the normal scope lookup. This exception also applies for calls that pass pointers or references to a class type.

In this example, when the compiler sees the “call” to `operator>>`, it looks for a matching function in the current scope, including the scopes enclosing the output statement. In addition, because the `>>` expression has parameters of class type, the compiler also looks in the namespace(s) in which the types of `cin` and `s` are defined. Thus, for this call, the compiler looks in the `std` namespace, which defines the `istream` and `string` types. When it searches `std`, the compiler finds the `string` output operator function.

This exception in the lookup rules allows nonmember functions that are conceptually part of the interface to a class to be used without requiring a separate `using` declaration. In the absence of this exception to the lookup rules, either we would have to provide an appropriate `using` declaration for the output operator:

```
using std::operator>>;           // needed to allow cin >> s
```

or we would have to use the function-call notation in order to include the namespace qualifier:

```
std::operator>>(std::cin, s);    // ok: explicitly use std:::>>
```

There would be no way to use operator syntax. Either of these declarations is awkward and would make simple uses of the IO library more complicated.

## Lookup and `std::move` and `std::forward`

Many, perhaps even most, C++ programmers never have to think about argument-dependent lookup. Ordinarily, if an application defines a name that is also defined in the library, one of two things is true: Either normal overloading determines (correctly) whether a particular call is intended for the application version or the one from the library, or the application never intends to use the library function.

Now consider the library `move` and `forward` functions. Both of these functions are template functions, and the library defines versions of them that have a single rvalue reference function parameter. As we've seen, in a function template, an rvalue reference parameter can match any type (§ 16.2.6, p. 690). If our application defines a function named `move` that takes a single parameter, then—no matter what type the parameter has—the application's version of `move` will collide with the library version. Similarly for `forward`.

As a result, name collisions with `move` (and `forward`) are more likely than collisions with other library functions. In addition, because `move` and `forward` do very specialized type manipulations, the chances that an application specifically wants to override the behavior of these functions are pretty small.

The fact that collisions are more likely—and are less likely to be intentional—explains why we suggest always using the fully qualified versions of these names (§ 12.1.5, p. 470). So long as we write `std::move` rather than `move`, we know that we will get the version from the standard library.

## Friend Declarations and Argument-Dependent Lookup



Recall that when a class declares a friend, the friend declaration does not make the friend visible (§ 7.2.1, p. 270). However, an otherwise undeclared class or function that is first named in a friend declaration is assumed to be a member of the closest enclosing namespace. The combination of this rule and argument-dependent lookup can lead to surprises:

```
namespace A {
    class C {
        // two friends; neither is declared apart from a friend declaration
        // these functions implicitly are members of namespace A
        friend void f2();           // won't be found, unless otherwise declared
        friend void f(const C&);   // found by argument-dependent lookup
    };
}
```

Here, both `f` and `f2` are members of namespace `A`. Through argument-dependent lookup, we can call `f` even if there is no additional declaration for `f`:

```
int main()
{
    A::C cobj;
    f(cobj);      // ok: finds A::f through the friend declaration in A::C
    f2();         // error: A::f2 not declared
}
```

Because `f` takes an argument of a class type, and `f` is implicitly declared in the same namespace as `C`, `f` is found when called. Because `f2` has no parameter, it will not be found.

### EXERCISES SECTION 18.2.3

**Exercise 18.18:** Given the following typical definition of `swap` § 13.3 (p. 517), determine which version of `swap` is used if `mem1` is a `string`. What if `mem1` is an `int`? Explain how name lookup works in both cases.

```
void swap(T v1, T v2)
{
    using std::swap;
    swap(v1.mem1, v2.mem1);
    // swap remaining members of type T
}
```

**Exercise 18.19:** What if the call to `swap` was `std::swap(v1.mem1, v2.mem1)`?

## 18.2.4 Overloading and Namespaces

Namespaces have two impacts on function matching (§ 6.4, p. 233). One of these should be obvious: A `using` declaration or directive can add functions to the candidate set. The other is much more subtle.



### Argument-Dependent Lookup and Overloading

As we saw in the previous section, name lookup for functions that have class-type arguments includes the namespace in which each argument's class is defined. This rule also impacts how we determine the candidate set. Each namespace that defines a class used as an argument (and those that define its base classes) is searched for candidate functions. Any functions in those namespaces that have the same name as the called function are added to the candidate set. These functions are added *even though they otherwise are not visible at the point of the call*:

```
namespace NS {
    class Quote { /* ... */ };
    void display(const Quote&) { /* ... */ }
}
// Bulk_item's base class is declared in namespace NS
class Bulk_item : public NS::Quote { /* ... */ };
int main() {
    Bulk_item book1;
    display(book1);
    return 0;
}
```

The argument we passed to `display` has class type `Bulk_item`. The candidate functions for the call to `display` are not only the functions with declarations that are in scope where `display` is called, but also the functions in the namespace where `Bulk_item` and its base class, `Quote`, are declared. The function `display(const Quote&)` declared in namespace `NS` is added to the set of candidate functions.

### Overloading and `using` Declarations

To understand the interaction between `using` declarations and overloading, it is important to remember that a `using` declaration declares a name, not a specific function (§ 15.6, p. 621):

```
using NS::print(int); // error: cannot specify a parameter list
using NS::print;      // ok: using declarations specify names only
```

When we write a `using` declaration for a function, all the versions of that function are brought into the current scope.

A `using` declaration incorporates all versions to ensure that the interface of the namespace is not violated. The author of a library provided different functions for a reason. Allowing users to selectively ignore some but not all of the functions from a set of overloaded functions could lead to surprising program behavior.

The functions introduced by a `using` declaration overload any other declarations of the functions with the same name already present in the scope where the `using` declaration appears. If the `using` declaration appears in a local scope, these names hide existing declarations for that name in the outer scope. If the `using` declaration introduces a function in a scope that already has a function of the same name with the same parameter list, then the `using` declaration is in error. Otherwise, the `using` declaration defines additional overloaded instances of the given name. The effect is to increase the set of candidate functions.

## Overloading and `using` Directives

A `using` directive lifts the namespace members into the enclosing scope. If a namespace function has the same name as a function declared in the scope at which the namespace is placed, then the namespace member is added to the overload set:

```
namespace libs_R_us {
    extern void print(int);
    extern void print(double);
}

// ordinary declaration
void print(const std::string &);

// this using directive adds names to the candidate set for calls to print:
using namespace libs_R_us;

// the candidates for calls to print at this point in the program are:
//   print(int) from libs_R_us
//   print(double) from libs_R_us
//   print(const std::string &) declared explicitly

void fooBar(int ival)
{
    print("Value: "); // calls global print(const string &)
    print(ival);      // calls libs_R_us::print(int)
}
```

Differently from how `using` declarations work, it is not an error if a `using` directive introduces a function that has the same parameters as an existing function. As with other conflicts generated by `using` directives, there is no problem unless we try to call the function without specifying whether we want the one from the namespace or from the current scope.

## Overloading across Multiple `using` Directives

If many `using` directives are present, then the names from each namespace become part of the candidate set:

```
namespace AW {
    int print(int);
}

namespace Primer {
    double print(double);
}
```

```
// using directives create an overload set of functions from different namespaces
using namespace AW;
using namespace Primer;
long double print(long double);
int main() {
    print(1);    // calls AW::print(int)
    print(3.1); // calls Primer::print(double)
    return 0;
}
```

The overload set for the function `print` in global scope contains the functions `print(int)`, `print(double)`, and `print(long double)`. These functions are all part of the overload set considered for the function calls in `main`, even though these functions were originally declared in different namespace scopes.

### EXERCISES SECTION 18.2.4

**Exercise 18.20:** In the following code, determine which function, if any, matches the call to `compute`. List the candidate and viable functions. What type conversions, if any, are 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);
void f()
{
    compute(0);
}
```

What would happen if the `using` declaration were located in `main` before the call to `compute`? Answer the same questions as before.

## 18.3 Multiple and Virtual Inheritance

**Multiple inheritance** is the ability to derive a class from more than one direct base class (§ 15.2.2, p. 600). A multiply derived class inherits the properties of all its parents. Although simple in concept, the details of intertwining multiple base classes can present tricky design-level and implementation-level problems.

To explore multiple inheritance, we'll use a pedagogical example of a zoo animal hierarchy. Our zoo animals exist at different levels of abstraction. There are the individual animals, distinguished by their names, such as Ling-ling, Mowgli, and Balou. Each animal belongs to a species; Ling-Ling, for example, is a giant

panda. Species, in turn, are members of families. A giant panda is a member of the bear family. Each family, in turn, is a member of the animal kingdom—in this case, the more limited kingdom of a particular zoo.

We'll define an abstract `ZooAnimal` class to hold information that is common to all the zoo animals and provides the most general interface. The `Bear` class will contain information that is unique to the `Bear` family, and so on.

In addition to the `ZooAnimal` classes, our application will contain auxiliary classes that encapsulate various abstractions such as endangered animals. In our implementation of a `Panda` class, for example, a `Panda` is multiply derived from `Bear` and `Endangered`.

### 18.3.1 Multiple Inheritance

The derivation list in a derived class can contain more than one base class:

```
class Bear : public ZooAnimal {  
    ...  
class Panda : public Bear, public Endangered { /* ... */ };
```

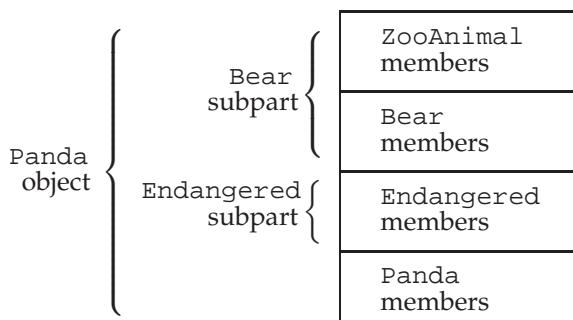
Each base class has an optional access specifier (§ 15.5, p. 612). As usual, if the access specifier is omitted, the specifier defaults to `private` if the `class` keyword is used and to `public` if `struct` is used (§ 15.5, p. 616).

As with single inheritance, the derivation list may include only classes that have been defined and that were not defined as `final` (§ 15.2.2, p. 600). There is no language-imposed limit on the number of base classes from which a class can be derived. A base class may appear only once in a given derivation list.

### Multiply Derived Classes Inherit State from Each Base Class

Under multiple inheritance, an object of a derived class contains a subobject for each of its base classes (§ 15.2.2, p. 597). For example, as illustrated in Figure 18.2, a `Panda` object has a `Bear` part (which itself contains a `ZooAnimal` part), an `Endangered` class part, and the `nonstatic` data members, if any, declared within the `Panda` class.

Figure 18.2: Conceptual Structure of a `Panda` Object



## Derived Constructors Initialize All Base Classes

Constructing an object of derived type constructs and initializes all its base sub-objects. As is the case for inheriting from a single base class (§ 15.2.2, p. 598), a derived type's constructor initializer may initialize only its direct base classes:

```
// explicitly initialize both base classes
Panda::Panda(std::string name, bool onExhibit)
    : Bear(name, onExhibit, "Panda"),
      Endangered(Endangered::critical) { }

// implicitly uses the Bear default constructor to initialize the Bear subobject
Panda::Panda()
    : Endangered(Endangered::critical) { }
```

The constructor initializer list may pass arguments to each of the direct base classes. The order in which base classes are constructed depends on the order in which they appear in the class derivation list. The order in which they appear in the constructor initializer list is irrelevant. A `Panda` object is initialized as follows:

- `ZooAnimal`, the ultimate base class up the hierarchy from `Panda`'s first direct base class, `Bear`, is initialized first.
- `Bear`, the first direct base class, is initialized next.
- `Endangered`, the second direct base, is initialized next.
- `Panda`, the most derived part, is initialized last.

## Inherited Constructors and Multiple Inheritance

 Under the new standard, a derived class can inherit its constructors from one or more of its base classes (§ 15.7.4, p. 628). It is an error to inherit the same constructor (i.e., one with the same parameter list) from more than one base class:

```
struct Base1 {
    Base1() = default;
    Base1(const std::string&);
    Base1(std::shared_ptr<int>);

};

struct Base2 {
    Base2() = default;
    Base2(const std::string&);
    Base2(int);

};

// error: D1 attempts to inherit D1::D1(const string&) from both base classes
struct D1: public Base1, public Base2 {
    using Base1::Base1; // inherit constructors from Base1
    using Base2::Base2; // inherit constructors from Base2
};
```

A class that inherits the same constructor from more than one base class must define its own version of that constructor:

```
struct D2: public Base1, public Base2 {  
    using Base1::Base1; // inherit constructors from Base1  
    using Base2::Base2; // inherit constructors from Base2  
    // D2 must define its own constructor that takes a string  
    D2(const string &s): Base1(s), Base2(s) {}  
    D2() = default; // needed once D2 defines its own constructor  
};
```

## Destructors and Multiple Inheritance

As usual, the destructor in a derived class is responsible for cleaning up resources allocated by that class only—the members and all the base class(es) of the derived class are automatically destroyed. The synthesized destructor has an empty function body.

Destructors are always invoked in the reverse order from which the constructors are run. In our example, the order in which the destructors are called is `~Panda, ~Endangered, ~Bear, ~ZooAnimal`.

## Copy and Move Operations for Multiply Derived Classes

As is the case for single inheritance, classes with multiple bases that define their own copy/move constructors and assignment operators must copy, move, or assign the whole object (§ 15.7.2, p. 623). The base parts of a multiply derived class are automatically copied, moved, or assigned only if the derived class uses the synthesized versions of these members. In the synthesized copy-control members, each base class is implicitly constructed, assigned, or destroyed, using the corresponding member from that base class.

For example, assuming that Panda uses the synthesized members, then the initialization of `ling_ling`:

```
Panda ying_yang("ying_yang");  
Panda ling_ling = ying_yang; // uses the copy constructor
```

will invoke the Bear copy constructor, which in turn runs the ZooAnimal copy constructor before executing the Bear copy constructor. Once the Bear portion of `ling_ling` is constructed, the Endangered copy constructor is run to create that part of the object. Finally, the Panda copy constructor is run. Similarly, for the synthesized move constructor.

The synthesized copy-assignment operator behaves similarly to the copy constructor. It assigns the Bear (and through Bear, the ZooAnimal) parts of the object first. Next, it assigns the Endangered part, and finally the Panda part. Move assignment behaves similarly.

### 18.3.2 Conversions and Multiple Base Classes

Under single inheritance, a pointer or a reference to a derived class can be converted automatically to a pointer or a reference to an accessible base class (§ 15.2.2, p. 597, and § 15.5, p. 613). The same holds true with multiple inheritance. A pointer or reference to any of an object's (accessible) base classes can be used to point or

### EXERCISES SECTION 18.3.1

**Exercise 18.21:** Explain the following declarations. Identify any that are in error and explain why they are incorrect:

- (a) class CADVehicle : public CAD, Vehicle { ... };
- (b) class DblList: public List, public List { ... };
- (c) class iostream: public istream, public ostream { ... };

**Exercise 18.22:** 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;
```

refer to a derived object. For example, a pointer or reference to ZooAnimal, Bear, or Endangered can be bound to a Panda object:

```
// operations that take references to base classes of type Panda
void print(const Bear&);
void highlight(const Endangered&);
ostream& operator<<(ostream&, const ZooAnimal&);

Panda ying_yang("ying_yang");

print(ying_yang);      // passes Panda to a reference to Bear
highlight(ying_yang); // passes Panda to a reference to Endangered
cout << ying_yang << endl; // passes Panda to a reference to ZooAnimal
```

The compiler makes no attempt to distinguish between base classes in terms of a derived-class conversion. Converting to each base class is equally good. For example, if there was an overloaded version of `print`:

```
void print(const Bear&);
void print(const Endangered&);
```

an unqualified call to `print` with a Panda object would be a compile-time error:

```
Panda ying_yang("ying_yang");
print(ying_yang);           // error: ambiguous
```

### Lookup Based on Type of Pointer or Reference

As with single inheritance, the static type of the object, pointer, or reference determines which members we can use (§ 15.6, p. 617). If we use a `ZooAnimal` pointer,

only the operations defined in that class are usable. The Bear-specific, Panda-specific, and Endangered portions of the Panda interface are invisible. Similarly, a Bear pointer or reference knows only about the Bear and ZooAnimal members; an Endangered pointer or reference is limited to the Endangered members.

As an example, consider the following calls, which assume that our classes define the virtual functions listed in Table 18.1:

```
Bear *pb = new Panda("ying_yang");
pb->print();           // ok: Panda::print()
pb->cuddle();          // error: not part of the Bear interface
pb->highlight();        // error: not part of the Bear interface
delete pb;              // ok: Panda::~Panda()
```

When a Panda is used via an Endangered pointer or reference, the Panda-specific and Bear portions of the Panda interface are invisible:

```
Endangered *pe = new Panda("ying_yang");
pe->print();           // ok: Panda::print()
pe->toes();             // error: not part of the Endangered interface
pe->cuddle();           // error: not part of the Endangered interface
pe->highlight();         // ok: Panda::highlight()
delete pe;               // ok: Panda::~Panda()
```

**Table 18.1: Virtual Functions in the ZooAnimal/Endangered Classes**

| Function   | Class Defining Own Version                                                   |
|------------|------------------------------------------------------------------------------|
| print      | ZooAnimal::ZooAnimal<br>Bear::Bear<br>Endangered::Endangered<br>Panda::Panda |
| highlight  | Endangered::Endangered<br>Panda::Panda                                       |
| toes       | Bear::Bear<br>Panda::Panda                                                   |
| cuddle     | Panda::Panda                                                                 |
| destructor | ZooAnimal::ZooAnimal<br>Endangered::Endangered                               |

### 18.3.3 Class Scope under Multiple Inheritance

Under single inheritance, the scope of a derived class is nested within the scope of its direct and indirect base classes (§ 15.6, p. 617). Lookup happens by searching up the inheritance hierarchy until the given name is found. Names defined in a derived class hide uses of that name inside a base.

Under multiple inheritance, this same lookup happens *simultaneously* among all the direct base classes. If a name is found through more than one base class, then use of that name is ambiguous.

## EXERCISES SECTION 18.3.2

**Exercise 18.23:** Using the hierarchy in exercise 18.22 along with class D defined below, and assuming each class defines a default constructor, which, if any, of the following conversions are not permitted?

```
class D : public X, public C { ... };
D *pd = new D;
(a) X *px = pd;      (b) A *pa = pd;
(c) B *pb = pd;      (d) C *pc = pd;
```

**Exercise 18.24:** On page 807 we presented a series of calls made through a Bear pointer that pointed to a Panda object. Explain each call assuming we used a ZooAnimal pointer pointing to a Panda object instead.

**Exercise 18.25:** 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;
```

In our example, if we use a name through a Panda object, pointer, or reference, both the Endangered and the Bear/ZooAnimal subtrees are examined in parallel. If the name is found in more than one subtree, then the use of the name is ambiguous. It is perfectly legal for a class to inherit multiple members with the same name. However, if we want to use that name, we must specify which version we want to use.



When a class has multiple base classes, it is possible for that derived class to inherit a member with the same name from two or more of its base classes. Unqualified uses of that name are ambiguous.

For example, if both ZooAnimal and Endangered define a member named max\_weight, and Panda does not define that member, this call is an error:

```
double d = ying_yang.max_weight();
```

The derivation of Panda, which results in Panda having two members named max\_weight, is perfectly legal. The derivation generates a *potential* ambiguity. That ambiguity is avoided if no Panda object ever calls max\_weight. The error

would also be avoided if each call to `max_weight` specifically indicated which version to run—`ZooAnimal::max_weight` or `Endangered::max_weight`. An error results only if there is an ambiguous attempt to use the member.

The ambiguity of the two inherited `max_weight` members is reasonably obvious. It might be more surprising to learn that an error would be generated even if the two inherited functions had different parameter lists. Similarly, it would be an error even if the `max_weight` function were `private` in one class and `public` or `protected` in the other. Finally, if `max_weight` were defined in `Bear` and not in `ZooAnimal`, the call would still be in error.

As always, name lookup happens before type checking (§ 6.4.1, p. 234). When the compiler finds `max_weight` in two different scopes, it generates an error noting that the call is ambiguous.

The best way to avoid potential ambiguities is to define a version of the function in the derived class that resolves the ambiguity. For example, we should give our `Panda` class a `max_weight` function that resolves the ambiguity:

```
double Panda::max_weight() const
{
    return std::max(ZooAnimal::max_weight(),
                    Endangered::max_weight());
}
```

### EXERCISES SECTION 18.3.3

**Exercise 18.26:** Given the hierarchy in the box on page 810, why is the following call to `print` an error? Revise MI to allow this call to `print` to compile and execute correctly.

```
MI mi;
mi.print(42);
```

**Exercise 18.27:** Given the class hierarchy in the box on page 810 and assuming we add a function named `foo` to MI as follows:

```
int ival;
double dval;
void MI::foo(double cval)
{
    int dval;
    // exercise questions occur here
}
```

- (a) List all the names visible from within `MI::foo`.
- (b) Are any names visible from more than one base class?
- (c) Assign to the local instance of `dval` the sum of the `dval` member of `Base1` and the `dval` member of `Derived`.
- (d) Assign the value of the last element in `MI::dvec` to `Base2::fval`.
- (e) Assign `cval` from `Base1` to the first character in `sval` from `Derived`.

**CODE FOR EXERCISES TO SECTION 18.3.3**

```

struct Basel {
    void print(int) const;           // public by default
protected:
    int      ival;
    double   dval;
    char     cval;
private:
    int      *id;
};

struct Base2 {
    void print(double) const;       // public by default
protected:
    double   fval;
private:
    double   dval;
};

struct Derived : public Basel {
    void print(std::string) const;   // public by default
protected:
    std::string sval;
    double      dval;
};

struct MI : public Derived, public Base2 {
    void print(std::vector<double>); // public by default
protected:
    int          *ival;
    std::vector<double>  dvec;
};

```

### 18.3.4 Virtual Inheritance

Although the derivation list of a class may not include the same base class more than once, a class can inherit from the same base class more than once. It might inherit the same base indirectly from two of its own direct base classes, or it might inherit a particular class directly and indirectly through another of its base classes.

As an example, the IO library `istream` and `ostream` classes each inherit from a common abstract base class named `basic_ios`. That class holds the stream's buffer and manages the stream's condition state. The class `iostream`, which can both read and write to a stream, inherits directly from both `istream` and `ostream`. Because both types inherit from `basic_ios`, `iostream` inherits that base class twice, once through `istream` and once through `ostream`.

By default, a derived object contains a separate subpart corresponding to each class in its derivation chain. If the same base class appears more than once in the derivation, then the derived object will have more than one subobject of that type.

This default doesn't work for a class such as `iostream`. An `iostream` object

wants to use the same buffer for both reading and writing, and it wants its condition state to reflect both input and output operations. If an `iostream` object has two copies of its `basic_ios` class, this sharing isn't possible.

In C++ we solve this kind of problem by using **virtual inheritance**. Virtual inheritance lets a class specify that it is willing to share its base class. The shared base-class subobject is called a **virtual base class**. Regardless of how often the same virtual base appears in an inheritance hierarchy, the derived object contains only one, shared subobject for that virtual base class.

## A Different Panda Class

In the past, there was some debate as to whether panda belongs to the raccoon or the bear family. To reflect this debate, we can change `Panda` to inherit from both `Bear` and `Raccoon`. To avoid giving `Panda` two `ZooAnimal` base parts, we'll define `Bear` and `Raccoon` to inherit virtually from `ZooAnimal`. Figure 18.3 illustrates our new hierarchy.

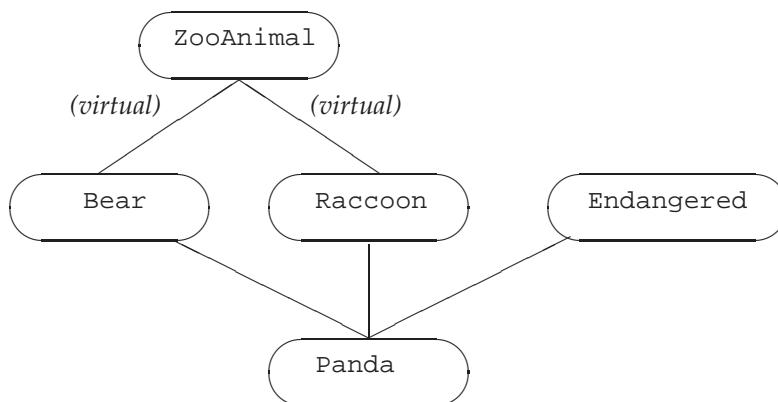
Looking at our new hierarchy, we'll notice a nonintuitive aspect of virtual inheritance. The virtual derivation has to be made before the need for it appears. For example, in our classes, the need for virtual inheritance arises only when we define `Panda`. However, if `Bear` and `Raccoon` had not specified `virtual` on their derivation from `ZooAnimal`, the designer of the `Panda` class would be out of luck.

In practice, the requirement that an intermediate base class specify its inheritance as virtual rarely causes any problems. Ordinarily, a class hierarchy that uses virtual inheritance is designed at one time either by one individual or by a single project design group. It is exceedingly rare for a class to be developed independently that needs a virtual base in one of its base classes and in which the developer of the new base class cannot change the existing hierarchy.



Virtual derivation affects the classes that subsequently derive from a class with a virtual base; it doesn't affect the derived class itself.

Figure 18.3: Virtual Inheritance Panda Hierarchy



## Using a Virtual Base Class

We specify that a base class is virtual by including the keyword `virtual` in the derivation list:

```
// the order of the keywords public and virtual is not significant
class Raccoon : public virtual ZooAnimal { /* ... */ };
class Bear : virtual public ZooAnimal { /* ... */ };
```

Here we've made `ZooAnimal` a virtual base class of both `Bear` and `Raccoon`.

The `virtual` specifier states a willingness to share a single instance of the named base class within a subsequently derived class. There are no special constraints on a class used as a virtual base class.

We do nothing special to inherit from a class that has a virtual base:

```
class Panda : public Bear,
              public Raccoon, public Endangered {
};
```

Here `Panda` inherits `ZooAnimal` through both its `Raccoon` and `Bear` base classes. However, because those classes inherited virtually from `ZooAnimal`, `Panda` has only one `ZooAnimal` base subpart.

## Normal Conversions to Base Are Supported

An object of a derived class can be manipulated (as usual) through a pointer or a reference to an accessible base-class type regardless of whether the base class is virtual. For example, all of the following `Panda` base-class conversions are legal:

```
void dance(const Bear&);
void rummage(const Raccoon&);
ostream& operator<<(ostream&, const ZooAnimal&);
Panda ying_yang;
dance(ying_yang); // ok: passes Panda object as a Bear
rummage(ying_yang); // ok: passes Panda object as a Raccoon
cout << ying_yang; // ok: passes Panda object as a ZooAnimal
```

## Visibility of Virtual Base-Class Members

Because there is only one shared subobject corresponding to each shared virtual base, members in that base can be accessed directly and unambiguously. Moreover, if a member from the virtual base is overridden along only one derivation path, then that overridden member can still be accessed directly. If the member is overridden by more than one base, then the derived class generally must define its own version as well.

For example, assume class `B` defines a member named `x`; class `D1` inherits virtually from `B` as does class `D2`; and class `D` inherits from `D1` and `D2`. From the scope of `D`, `x` is visible through both of its base classes. If we use `x` through a `D` object, there are three possibilities:

- If `x` is not defined in either `D1` or `D2` it will be resolved as a member in `B`; there is no ambiguity. A `D` object contains only one instance of `x`.

- If *x* is a member of *B* and also a member in one, but not both, of *D1* and *D2*, there is again no ambiguity—the version in the derived class is given precedence over the shared virtual base class, *B*.
- If *x* is defined in both *D1* and *D2*, then direct access to that member is ambiguous.

As in a nonvirtual multiple inheritance hierarchy, ambiguities of this sort are best resolved by the derived class providing its own instance of that member.

### EXERCISES SECTION 18.3.4

**Exercise 18.28:** Given the following class hierarchy, which inherited members can be accessed without qualification from within the *VMI* class? Which require qualification? Explain your reasoning.

```
struct Base {  
    void bar(int); // public by default  
protected:  
    int ival;  
};  
struct Derived1 : virtual public Base {  
    void bar(char); // public by default  
    void foo(char);  
protected:  
    char cval;  
};  
struct Derived2 : virtual public Base {  
    void foo(int); // public by default  
protected:  
    int ival;  
    char cval;  
};  
class VMI : public Derived1, public Derived2 { };
```

### 18.3.5 Constructors and Virtual Inheritance

In a virtual derivation, the virtual base is initialized by the *most derived constructor*. In our example, when we create a *Panda* object, the *Panda* constructor alone controls how the *ZooAnimal* base class is initialized.

To understand this rule, consider what would happen if normal initialization rules applied. In that case, a virtual base class might be initialized more than once. It would be initialized along each inheritance path that contains that virtual base. In our *ZooAnimal* example, if normal initialization rules applied, both *Bear* and *Raccoon* would initialize the *ZooAnimal* part of a *Panda* object.

Of course, each class in the hierarchy might at some point be the “most derived” object. As long as we can create independent objects of a type derived from

a virtual base, the constructors in that class must initialize its virtual base. For example, in our hierarchy, when a Bear (or a Raccoon) object is created, there is no further derived type involved. In this case, the Bear (or Raccoon) constructors directly initialize their ZooAnimal base as usual:

```
Bear::Bear(std::string name, bool onExhibit):
    ZooAnimal(name, onExhibit, "Bear") { }
Raccoon::Raccoon(std::string name, bool onExhibit)
    : ZooAnimal(name, onExhibit, "Raccoon") { }
```

When a Panda is created, it is the most derived type and controls initialization of the shared ZooAnimal base. Even though ZooAnimal is not a direct base of Panda, the Panda constructor initializes ZooAnimal:

```
Panda::Panda(std::string name, bool onExhibit)
    : ZooAnimal(name, onExhibit, "Panda"),
      Bear(name, onExhibit),
      Raccoon(name, onExhibit),
      Endangered(Endangered::critical),
      sleeping_flag(false) { }
```

## How a Virtually Inherited Object Is Constructed

The construction order for an object with a virtual base is slightly modified from the normal order: The virtual base subparts of the object are initialized first, using initializers provided in the constructor for the most derived class. Once the virtual base subparts of the object are constructed, the direct base subparts are constructed in the order in which they appear in the derivation list.

For example, when a Panda object is created:

- The (virtual base class) ZooAnimal part is constructed first, using the initializers specified in the Panda constructor initializer list.
- The Bear part is constructed next.
- The Raccoon part is constructed next.
- The third direct base, Endangered, is constructed next.
- Finally, the Panda part is constructed.

If the Panda constructor does not explicitly initialize the ZooAnimal base class, then the ZooAnimal default constructor is used. If ZooAnimal doesn't have a default constructor, then the code is in error.



Virtual base classes are always constructed prior to nonvirtual base classes regardless of where they appear in the inheritance hierarchy.

## Constructor and Destructor Order

A class can have more than one virtual base class. In that case, the virtual subobjects are constructed in left-to-right order as they appear in the derivation list. For

example, in the following whimsical `TeddyBear` derivation, there are two virtual base classes: `ToyAnimal`, a direct virtual base, and `ZooAnimal`, which is a virtual base class of `Bear`:

```
class Character { /* ... */ };
class BookCharacter : public Character { /* ... */ };
class ToyAnimal { /* ... */ };
class TeddyBear : public BookCharacter,
                  public Bear, public virtual ToyAnimal
{ /* ... */ };
```

The direct base classes are examined in declaration order to determine whether there are any virtual base classes. If so, the virtual bases are constructed first, followed by the nonvirtual base-class constructors in declaration order. Thus, to create a `TeddyBear`, the constructors are invoked in the following order:

```
ZooAnimal();           // Bear's virtual base class
ToyAnimal();           // direct virtual base class
Character();          // indirect base class of first nonvirtual base class
BookCharacter();       // first direct nonvirtual base class
Bear();                // second direct nonvirtual base class
TeddyBear();           // most derived class
```

The same order is used in the synthesized copy and move constructors, and members are assigned in this order in the synthesized assignment operators. As usual, an object is destroyed in reverse order from which it was constructed. The `TeddyBear` part will be destroyed first and the `ZooAnimal` part last.

### EXERCISES SECTION 18.3.5

**Exercise 18.29:** Given the following class hierarchy:

```
class Class { ... };
class Base : public Class { ... };
class D1 : virtual public Base { ... };
class D2 : virtual public Base { ... };
class MI : public D1, public D2 { ... };
class Final : public MI, public Class { ... };
```

- (a) In what order are constructors and destructors run on a `Final` object?
- (b) A `Final` object has how many `Base` parts? How many `Class` parts?
- (c) Which of the following assignments is a compile-time error?

```
Base *pb;      Class *pc;      MI *pmi;      D2 *pd2;
(a) pb = new Class;    (b) pc = new Final;
(c) pmi = pb;        (d) pd2 = pmi;
```

**Exercise 18.30:** Define a default constructor, a copy constructor, and a constructor that has an `int` parameter in `Base`. Define the same three constructors in each derived class. Each constructor should use its argument to initialize its `Base` part.

C++ is intended for use in a wide variety of applications. As a result, it contains features that are particular to some applications and that need never be used by others. In this chapter we look at some of the less-commonly used features in the language.

## 19.1 Controlling Memory Allocation

Some applications have specialized memory allocation needs that cannot be met by the standard memory management facilities. Such applications need to take over the details of how memory is allocated, for example, by arranging for new to put objects into particular kinds of memory. To do so, they can overload the new and delete operators to control memory allocation.

### 19.1.1 Overloading new and delete

Although we say that we can “overload new and delete,” overloading these operators is quite different from the way we overload other operators. In order to understand how we overload these operators, we first need to know a bit more about how new and delete expressions work.

When we use a new expression:

```
// new expressions
string *sp = new string("a value"); // allocate and initialize a string
string *arr = new string[10]; // allocate ten default initialized strings
```

three steps actually happen. First, the expression calls a library function named **operator new** (or **operator new[ ]**). This function allocates raw, untyped memory large enough to hold an object (or an array of objects) of the specified type. Next, the compiler runs the appropriate constructor to construct the object(s) from the specified initializers. Finally, a pointer to the newly allocated and constructed object is returned.

When we use a delete expression to delete a dynamically allocated object:

```
delete sp; // destroy *sp and free the memory to which sp points
delete [] arr; // destroy the elements in the array and free the memory
```

two steps happen. First, the appropriate destructor is run on the object to which `sp` points or on the elements in the array to which `arr` points. Next, the compiler frees the memory by calling a library function named **operator delete** or **operator delete[ ]**, respectively.

Applications that want to take control of memory allocation define their own versions of the **operator new** and **operator delete** functions. Even though the library contains definitions for these functions, we can define our own versions of them and the compiler won’t complain about duplicate definitions. Instead, the compiler will use our version in place of the one defined by the library.



When we define the global operator new and operator delete functions, we take over responsibility for all dynamic memory allocation. These functions *must* be correct: They form a vital part of all processing in the program.

Applications can define operator new and operator delete functions in the global scope and/or as member functions. When the compiler sees a new or delete expression, it looks for the corresponding operator function to call. If the object being allocated (deallocated) has class type, the compiler first looks in the scope of the class, including any base classes. If the class has a member operator new or operator delete, that function is used by the new or delete expression. Otherwise, the compiler looks for a matching function in the global scope. If the compiler finds a user-defined version, it uses that function to execute the new or delete expression. Otherwise, the standard library version is used.

We can use the scope operator to force a new or delete expression to bypass a class-specific function and use the one from the global scope. For example, `::new` will look only in the global scope for a matching operator new function. Similarly for `::delete`.

## The operator new and operator delete Interface

The library defines eight overloaded versions of operator new and delete functions. The first four support the versions of new that can throw a `bad_alloc` exception. The next four support nonthrowing versions of new:

```
// these versions might throw an exception
void *operator new(size_t);                                // allocate an object
void *operator new[](size_t);                             // allocate an array
void *operator delete(void*) noexcept;                  // free an object
void *operator delete[](void*) noexcept; // free an array

// versions that promise not to throw; see § 12.1.2 (p. 460)
void *operator new(size_t, nothrow_t&) noexcept;
void *operator new[](size_t, nothrow_t&) noexcept;
void *operator delete(void*, nothrow_t&) noexcept;
void *operator delete[](void*, nothrow_t&) noexcept;
```

The type `nothrow_t` is a struct defined in the new header. This type has no members. The new header also defines a const object named `nothrow`, which users can pass to signal they want the nonthrowing version of new (§ 12.1.2, p. 460). Like destructors, an operator delete must not throw an exception (§ 18.1.1, p. 774). When we overload these operators, we must specify that they will not throw, which we do through the `noexcept` exception specifier (§ 18.1.4, p. 779).

An application can define its own version of any of these functions. If it does so, it must define these functions in the global scope or as members of a class. When defined as members of a class, these operator functions are implicitly static (§ 7.6, p. 302). There is no need to declare them static explicitly, although it is legal to do so. The member new and delete functions must be static because they are used either before the object is constructed (operator new) or after it has been

destroyed (`operator delete`). There are, therefore, no member data for these functions to manipulate.

An `operator new` or `operator new[ ]` function must have a return type of `void*` and its first parameter must have type `size_t`. That parameter may not have a default argument. The `operator new` function is used when we allocate an object; `operator new[ ]` is called when we allocate an array. When the compiler calls `operator new`, it initializes the `size_t` parameter with the number of bytes required to hold an object of the specified type; when it calls `operator new[ ]`, it passes the number of bytes required to store an array of the given number of elements.

When we define our own `operator new` function, we can define additional parameters. A new expression that uses such functions must use the placement form of `new` (§ 12.1.2, p. 460) to pass arguments to these additional parameters. Although generally we may define our version of `operator new` to have whatever parameters are needed, we may not define a function with the following form:

```
void *operator new(size_t, void*); // this version may not be redefined
```

This specific form is reserved for use by the library and may not be redefined.

An `operator delete` or `operator delete[ ]` function must have a `void` return type and a first parameter of type `void*`. Executing a `delete` expression calls the appropriate `operator` function and initializes its `void*` parameter with a pointer to the memory to free.

When `operator delete` or `operator delete[ ]` is defined as a class member, the function may have a second parameter of type `size_t`. If present, the additional parameter is initialized with the size in bytes of the object addressed by the first parameter. The `size_t` parameter is used when we delete objects that are part of an inheritance hierarchy. If the base class has a virtual destructor (§ 15.7.1, p. 622), then the size passed to `operator delete` will vary depending on the dynamic type of the object to which the deleted pointer points. Moreover, the version of the `operator delete` function that is run will be the one from the dynamic type of the object.

#### TERMINOLOGY: NEW EXPRESSION VERSUS OPERATOR NEW FUNCTION

The library functions `operator new` and `operator delete` are misleadingly named. Unlike other `operator` functions, such as `operator=`, these functions do not overload the `new` or `delete` expressions. In fact, we cannot redefine the behavior of the `new` and `delete` expressions.

A `new` expression always executes by calling an `operator new` function to obtain memory and then constructing an object in that memory. A `delete` expression always executes by destroying an object and then calling an `operator delete` function to free the memory used by the object.

By providing our own definitions of the `operator new` and `operator delete` functions, we can change how memory is allocated. However, we cannot change this basic meaning of the `new` and `delete` operators.

## The `malloc` and `free` Functions

If you define your own global operator new and operator delete, those functions must allocate and deallocate memory somehow. Even if you define these functions in order to use a specialized memory allocator, it can still be useful for testing purposes to be able to allocate memory similarly to how the implementation normally does so.

To this end, we can use functions named `malloc` and `free` that C++ inherits from C. These functions, are defined in `cstdlib`.

The `malloc` function takes a `size_t` that says how many bytes to allocate. It returns a pointer to the memory that it allocated, or 0 if it was unable to allocate the memory. The `free` function takes a `void*` that is a copy of a pointer that was returned from `malloc` and returns the associated memory to the system. Calling `free(0)` has no effect.

A simple way to write operator new and operator delete is as follows:

```
void *operator new(size_t size) {
    if (void *mem = malloc(size))
        return mem;
    else
        throw bad_alloc();
}
void operator delete(void *mem) noexcept { free(mem); }
```

and similarly for the other versions of operator new and operator delete.

### EXERCISES SECTION 19.1.1

**Exercise 19.1:** Write your own operator `new(size_t)` function using `malloc` and use `free` to write the operator `delete(void*)` function.

**Exercise 19.2:** By default, the allocator class uses operator new to obtain storage and operator delete to free it. Recompile and rerun your StrVec programs (§ 13.5, p. 526) using your versions of the functions from the previous exercise.

## 19.1.2 Placement new Expressions

Although the operator new and operator delete functions are intended to be used by new expressions, they are ordinary functions in the library. As a result, ordinary code can call these functions directly.

In earlier versions of the language—before the allocator (§ 12.2.2, p. 481) class was part of the library—applications that wanted to separate allocation from initialization did so by calling operator new and operator delete. These functions behave analogously to the allocate and deallocate members of allocator. Like those members, operator new and operator delete functions allocate and deallocate memory but do not construct or destroy objects.

Differently from an allocator, there is no construct function we can call to construct objects in memory allocated by operator new. Instead, we use the **placement new** form of new (§ 12.1.2, p. 460) to construct an object. As we've seen, this form of new provides extra information to the allocation function. We can use placement new to pass an address, in which case the placement new expression has the form

```
new (place_address) type
new (place_address) type (initializers)
new (place_address) type [size]
new (place_address) type [size] { braced initializer list }
```

where *place\_address* must be a pointer and the *initializers* provide (a possibly empty) comma-separated list of initializers to use to construct the newly allocated object.

When called with an address and no other arguments, placement new uses operator new(*size\_t*, *void\**) to "allocate" its memory. This is the version of operator new that we are not allowed to redefine (§ 19.1.1, p. 822). This function does *not* allocate any memory; it simply returns its pointer argument. The overall new expression then finishes its work by initializing an object at the given address. In effect, placement new allows us to construct an object at a specific, preallocated memory address.



When passed a single argument that is a pointer, a placement new expression constructs an object but does not allocate memory.

Although in many ways using placement new is analogous to the construct member of an allocator, there is one important difference. The pointer that we pass to construct must point to space allocated by the same allocator object. The pointer that we pass to placement new need not point to memory allocated by operator new. Indeed, as we'll see in § 19.6 (p. 851), the pointer passed to a placement new expression need not even refer to dynamic memory.

## Explicit Destructor Invocation

Just as placement new is analogous to using `allocate`, an explicit call to a destructor is analogous to calling `destroy`. We call a destructor the same way we call any other member function on an object or through a pointer or reference to an object:

```
string *sp = new string("a value"); // allocate and initialize a string
sp->~string();
```

Here we invoke a destructor directly. The arrow operator dereferences the pointer *sp* to obtain the object to which *sp* points. We then call the destructor, which is the name of the type preceded by a tilde (~).

Like calling `destroy`, calling a destructor cleans up the given object but does not free the space in which that object resides. We can reuse the space if desired.



Calling a destructor destroys an object but does not free the memory.

## 19.2 Run-Time Type Identification

Run-time type identification (RTTI) is provided through two operators:

- The `typeid` operator, which returns the type of a given expression
- The `dynamic_cast` operator, which safely converts a pointer or reference to a base type into a pointer or reference to a derived type

When applied to pointers or references to types that have virtual functions, these operators use the dynamic type (§ 15.2.3, p. 601) of the object to which the pointer or reference is bound.

These operators are useful when we have a derived operation that we want to perform through a pointer or reference to a base-class object and it is not possible to make that operation a virtual function. Ordinarily, we should use virtual functions if we can. When the operation is virtual, the compiler automatically selects the right function according to the dynamic type of the object.

However, it is not always possible to define a virtual. If we cannot use a virtual, we can use one of the RTTI operators. On the other hand, using these operators is more error-prone than using virtual member functions: The programmer must *know* to which type the object should be cast and must check that the cast was performed successfully.



RTTI should be used with caution. When possible, it is better to define a virtual function rather than to take over managing the types directly.

### 19.2.1 The `dynamic_cast` Operator

A `dynamic_cast` has the following form:

```
dynamic_cast<type*>(e)
dynamic_cast<type&>(e)
dynamic_cast<type&&>(e)
```

where *type* must be a class type and (ordinarily) names a class that has virtual functions. In the first case, *e* must be a valid pointer (§ 2.3.2, p. 52); in the second, *e* must be an lvalue; and in the third, *e* must not be an lvalue.

In all cases, the type of *e* must be either a class type that is publicly derived from the target *type*, a `public` base class of the target *type*, or the same as the target *type*. If *e* has one of these types, then the cast will succeed. Otherwise, the cast fails. If a `dynamic_cast` to a pointer type fails, the result is 0. If a `dynamic_cast` to a reference type fails, the operator throws an exception of type `bad_cast`.

#### Pointer-Type `dynamic_casts`

As a simple example, assume that `Base` is a class with at least one virtual function and that class `Derived` is publicly derived from `Base`. If we have a pointer to `Base` named `bp`, we can cast it, at run time, to a pointer to `Derived` as follows:

```

if (Derived *dp = dynamic_cast<Derived*>(bp))
{
    // use the Derived object to which dp points
} else { // bp points at a Base object
    // use the Base object to which bp points
}

```

If `bp` points to a `Derived` object, then the cast will initialize `dp` to point to the `Derived` object to which `bp` points. In this case, it is safe for the code inside the `if` to use `Derived` operations. Otherwise, the result of the cast is 0. If `dp` is 0, the condition in the `if` fails. In this case, the `else` clause does processing appropriate to `Base` instead.



We can do a `dynamic_cast` on a null pointer; the result is a null pointer of the requested type.

It is worth noting that we defined `dp` inside the condition. By defining the variable in a condition, we do the cast and corresponding check as a single operation. Moreover, the pointer `dp` is not accessible outside the `if`. If the cast fails, then the unbound pointer is not available for use in subsequent code where we might forget to check whether the cast succeeded.



Performing a `dynamic_cast` in a condition ensures that the cast and test of its result are done in a single expression.

## Reference-Type `dynamic_cast`s

A `dynamic_cast` to a reference type differs from a `dynamic_cast` to a pointer type in how it signals that an error occurred. Because there is no such thing as a null reference, it is not possible to use the same error-reporting strategy for references that is used for pointers. When a cast to a reference type fails, the cast throws a `std::bad_cast` exception, which is defined in the `typeinfo` library header.

We can rewrite the previous example to use references as follows:

```

void f(const Base &b)
{
    try {
        const Derived &d = dynamic_cast<const Derived&>(b);
        // use the Derived object to which b referred
    } catch (bad_cast) {
        // handle the fact that the cast failed
    }
}

```

### 19.2.2 The `typeid` Operator

The second operator provided for RTTI is the **`typeid` operator**. The `typeid` operator allows a program to ask of an expression: What type is your object?

**EXERCISES SECTION 19.2.1**

**Exercise 19.3:** 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\_casts 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);

**Exercise 19.4:** Using the classes defined in the first exercise, rewrite the following code to convert the expression \*pa to the type C&:

```
if (C *pc = dynamic_cast< C*>(pa))
    // use C's members
} else {
    // use A's members
}
```

**Exercise 19.5:** When should you use a dynamic\_cast instead of a virtual function?

A typeid expression has the form typeid(e) where e is any expression or a type name. The result of a typeid operation is a reference to a const object of a library type named type\_info, or a type publicly derived from type\_info. § 19.2.4 (p. 831) covers this type in more detail. The type\_info class is defined in the typeinfo header.

The typeid operator can be used with expressions of any type. As usual, top-level const (§ 2.4.3, p. 63) is ignored, and if the expression is a reference, typeid returns the type to which the reference refers. When applied to an array or function, however, the standard conversion to pointer (§ 4.11.2, p. 161) is not done. That is, if we take typeid(a) and a is an array, the result describes an array type, not a pointer type.

When the operand is not of class type or is a class without virtual functions, then the typeid operator indicates the static type of the operand. When the operand is an lvalue of a class type that defines at least one virtual function, then the type is evaluated at run time.

## Using the typeid Operator

Ordinarily, we use typeid to compare the types of two expressions or to compare the type of an expression to a specified type:

```

Derived *dp = new Derived;
Base *bp = dp; // both pointers point to a Derived object
// compare the type of two objects at run time
if (typeid(*bp) == typeid(*dp)) {
    // bp and dp point to objects of the same type
}
// test whether the run-time type is a specific type
if (typeid(*bp) == typeid(Derived)) {
    // bp actually points to a Derived
}

```

In the first `if`, we compare the dynamic types of the objects to which `bp` and `dp` point. If both point to the same type, then the condition succeeds. Similarly, the second `if` succeeds if `bp` currently points to a `Derived` object.

Note that the operands to the `typeid` are objects—we used `*bp`, not `bp`:

```

// test always fails: the type of bp is pointer to Base
if (typeid(bp) == typeid(Derived)) {
    // code never executed
}

```

This condition compares the type `Base*` to type `Derived`. Although the pointer *points* at an object of class type that has virtual functions, the pointer *itself* is not a class-type object. The type `Base*` can be, and is, evaluated at compile time. That type is unequal to `Derived`, so the condition will always fail *regardless of the type of the object to which bp points*.



The `typeid` of a pointer (as opposed to the object to which the pointer points) returns the static, compile-time type of the pointer.

Whether `typeid` requires a run-time check determines whether the expression is evaluated. The compiler evaluates the expression only if the type has virtual functions. If the type has no `virtuals`, then `typeid` returns the static type of the expression; the compiler knows the static type without evaluating the expression.

If the dynamic type of the expression might differ from the static type, then the expression must be evaluated (at run time) to determine the resulting type. The distinction matters when we evaluate `typeid(*p)`. If `p` is a pointer to a type that does not have virtual functions, then `p` does not need to be a valid pointer. Otherwise, `*p` is evaluated at run time, in which case `p` must be a valid pointer. If `p` is a null pointer, then `typeid(*p)` throws a `bad_typeid` exception.

### 19.2.3 Using RTTI

As an example of when RTTI might be useful, consider a class hierarchy for which we'd like to implement the equality operator (§ 14.3.1, p. 561). Two objects are equal if they have the same type and same value for a given set of their data members. Each derived type may add its own data, which we will want to include when we test for equality.

**EXERCISES SECTION 19.2.2**

**Exercise 19.6:** Write an expression to dynamically cast a pointer to a `Query_base` to a pointer to an `AndQuery` (§ 15.9.1, p. 636). Test the cast by using objects of `AndQuery` and of another query type. Print a statement indicating whether the cast works and be sure that the output matches your expectations.

**Exercise 19.7:** Write the same cast, but cast a `Query_base` object to a reference to `AndQuery`. Repeat the test to ensure that your cast works correctly.

**Exercise 19.8:** Write a `typeid` expression to see whether two `Query_base` pointers point to the same type. Now check whether that type is an `AndQuery`.

We might think we could solve this problem by defining a set of virtual functions that would perform the equality test at each level in the hierarchy. Given those virtuals, we would define a single equality operator that operates on references to the base type. That operator could delegate its work to a virtual `equal` operation that would do the real work.

Unfortunately, this strategy doesn't quite work. Virtual functions must have the same parameter type(s) in both the base and derived classes (§ 15.3, p. 605). If we wanted to define a virtual `equal` function, that function must have a parameter that is a reference to the base class. If the parameter is a reference to base, the `equal` function could use only members from the base class. `equal` would have no way to compare members that are in the derived class but not in the base.

We can write our equality operation by realizing that the equality operator ought to return `false` if we attempt to compare objects of differing type. For example, if we try to compare a object of the base-class type with an object of a derived type, the `==` operator should return `false`.

Given this observation, we can now see that we can use RTTI to solve our problem. We'll define an equality operator whose parameters are references to the base-class type. The equality operator will use `typeid` to verify that the operands have the same type. If the operands differ, the `==` will return `false`. Otherwise, it will call a virtual `equal` function. Each class will define `equal` to compare the data elements of its own type. These operators will take a `Base&` parameter but will cast the operand to its own type before doing the comparison.

## The Class Hierarchy

To make the concept a bit more concrete, we'll define the following classes:

```
class Base {  
    friend bool operator==(const Base&, const Base&);  
public:  
    // interface members for Base  
protected:  
    virtual bool equal(const Base&) const;  
    // data and other implementation members of Base  
};
```

```

class Derived: public Base {
public:
    // other interface members for Derived
protected:
    bool equal(const Base&) const;
    // data and other implementation members of Derived
};

```

## A Type-Sensitive Equality Operator

Next let's look at how we might define the overall equality operator:

```

bool operator==(const Base &lhs, const Base &rhs)
{
    // returns false if typeids are different; otherwise makes a virtual call to equal
    return typeid(lhs) == typeid(rhs) && lhs.equal(rhs);
}

```

This operator returns `false` if the operands are different types. If they are the same type, then it delegates the real work of comparing the operands to the (virtual) `equal` function. If the operands are `Base` objects, then `Base::equal` will be called. If they are `Derived` objects, `Derived::equal` is called.

## The Virtual `equal` Functions

Each class in the hierarchy must define its own version of `equal`. All of the functions in the derived classes will start the same way: They'll cast their argument to the type of the class itself:

```

bool Derived::equal(const Base &rhs) const
{
    // we know the types are equal, so the cast won't throw
    auto r = dynamic_cast<const Derived&>(rhs);
    // do the work to compare two Derived objects and return the result
}

```

The cast should always succeed—after all, the function is called from the equality operator only after testing that the two operands are the same type. However, the cast is necessary so that the function can access the derived members of the right-hand operand.

## The Base-Class `equal` Function

This operation is a bit simpler than the others:

```

bool Base::equal(const Base &rhs) const
{
    // do whatever is required to compare to Base objects
}

```

There is no need to cast the parameter before using it. Both `*this` and the parameter are `Base` objects, so all the operations available for this object are also defined for the parameter type.

### 19.2.4 The `type_info` Class

The exact definition of the `type_info` class varies by compiler. However, the standard guarantees that the class will be defined in the `typeinfo` header and that the class will provide at least the operations listed in Table 19.1.

The class also provides a public virtual destructor, because it is intended to serve as a base class. When a compiler wants to provide additional type information, it normally does so in a class derived from `type_info`.

**Table 19.1: Operations on `type_info`**

|                            |                                                                                                                                                                      |
|----------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>t1 == t2</code>      | Returns <code>true</code> if the <code>type_info</code> objects <code>t1</code> and <code>t2</code> refer to the same type, <code>false</code> otherwise.            |
| <code>t1 != t2</code>      | Returns <code>true</code> if the <code>type_info</code> objects <code>t1</code> and <code>t2</code> refer to different types, <code>false</code> otherwise.          |
| <code>t.name()</code>      | Returns a C-style character string that is a printable version of the type name. Type names are generated in a system-dependent way.                                 |
| <code>t1.before(t2)</code> | Returns a <code>bool</code> that indicates whether <code>t1</code> comes before <code>t2</code> . The ordering imposed by <code>before</code> is compiler dependent. |

There is no `type_info` default constructor, and the copy and move constructors and the assignment operators are all defined as deleted (§ 13.1.6, p. 507). Therefore, we cannot define, copy, or assign objects of type `type_info`. The only way to create a `type_info` object is through the `typeid` operator.

The `name` member function returns a C-style character string for the name of the type represented by the `type_info` object. The value used for a given type depends on the compiler and in particular is not required to match the type names as used in a program. The only guarantee we have about the return from `name` is that it returns a unique string for each type. For example:

```
int arr[10];
Derived d;
Base *p = &d;

cout << typeid(42).name() << ", "
    << typeid(arr).name() << ", "
    << typeid(Sales_data).name() << ", "
    << typeid(std::string).name() << ", "
    << typeid(p).name() << ", "
    << typeid(*p).name() << endl;
```

This program, when executed on our machine, generates the following output:

```
i, A10_i, 10Sales_data, ss, P4Base, 7Derived
```



The `type_info` class varies by compiler. Some compilers provide additional member functions that provide additional information about types used in a program. You should consult the reference manual for your compiler to understand the exact `type_info` support provided.

## EXERCISES SECTION 19.2.4

**Exercise 19.9:** Write a program similar to the last one in this section to print the names your compiler uses for common type names. If your compiler gives output similar to ours, write a function that will translate those strings to more human-friendly form.

**Exercise 19.10:** 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;
```

## 19.3 Enumerations

Enumerations let us group together sets of integral constants. Like classes, each enumeration defines a new type. Enumerations are literal types (§ 7.5.6, p. 299).

C++ has two kinds of enumerations: scoped and unscoped. The new standard introduced **scoped enumerations**. We define a scoped enumeration using the keywords `enum class` (or, equivalently, `enum struct`), followed by the enumeration name and a comma-separated list of **enumerators** enclosed in curly braces. A semicolon follows the close curly:

```
enum class open_modes {input, output, append};
```

Here we defined an enumeration type named `open_modes` that has three enumerators: `input`, `output`, and `append`.

We define an **unscoped enumeration** by omitting the `class` (or `struct`) keyword. The enumeration name is optional in an unscoped enum:

```
enum color {red, yellow, green};           // unscoped enumeration
// unnamed, unscoped enum
enum {floatPrec = 6, doublePrec = 10, double_doublePrec = 10};
```

If the `enum` is unnamed, we may define objects of that type only as part of the `enum` definition. As with a class definition, we can provide a comma-separated list of declarators between the close curly and the semicolon that ends the `enum` definition (§ 2.6.1, p. 73).

## Enumerators

The names of the enumerators in a scoped enumeration follow normal scoping rules and are inaccessible outside the scope of the enumeration. The enumerator names in an unscoped enumeration are placed into the same scope as the enumeration itself:

```
enum color {red, yellow, green};      // unscoped enumeration
enum stoplight {red, yellow, green}; // error: redefines enumerators
enum class peppers {red, yellow, green}; // ok: enumerators are hidden
color eyes = green; // ok: enumerators are in scope for an unscoped enumeration
peppers p = green; // error: enumerators from peppers are not in scope
                      //           color::green is in scope but has the wrong type
color hair = color::red; // ok: we can explicitly access the enumerators
peppers p2 = peppers::red; // ok: using red from peppers
```

By default, enumerator values start at 0 and each enumerator has a value 1 greater than the preceding one. However, we can also supply initializers for one or more enumerators:

```
enum class intTypes {
    charTyp = 8, shortTyp = 16, intTyp = 16,
    longTyp = 32, long_longTyp = 64
};
```

As we see with the enumerators for `intTyp` and `shortTyp`, an enumerator value need not be unique. When we omit an initializer, the enumerator has a value 1 greater than the preceding enumerator.

Enumerators are `const` and, if initialized, their initializers must be constant expressions (§ 2.4.4, p. 65). Consequently, each enumerator is itself a constant expression. Because the enumerators are constant expressions, we can use them where a constant expression is required. For example, we can define `constexpr` variables of enumeration type:

```
constexpr intTypes charbits = intTypes::charTyp;
```

Similarly, we can use an `enum` as the expression in a `switch` statement and use the value of its enumerators as the case labels (§ 5.3.2, p. 178). For the same reason, we can also use an enumeration type as a nontype template parameter (§ 16.1.1, p. 654) and can initialize class `static` data members of enumeration type inside the class definition (§ 7.6, p. 302).

## Like Classes, Enumerations Define New Types

So long as the `enum` is named, we can define and initialize objects of that type. An `enum` object may be initialized or assigned only by one of its enumerators or by another object of the same `enum` type:

```
open_modes om = 2;      // error: 2 is not of type open_modes
om = open_modes::input; // ok: input is an enumerator of open_modes
```

Objects or enumerators of an unscoped enumeration type are automatically converted to an integral type. As a result, they can be used where an integral value is required:

```
int i = color::red;    // ok: unscoped enumerator implicitly converted to int
int j = peppers::red; // error: scoped enumerations are not implicitly converted
```

## Specifying the Size of an enum

Although each enum defines a unique type, it is represented by one of the built-in integral types. Under the new standard, we may specify that type by following the enum name with a colon and the name of the type we want to use:

```
C++ 11
enum intValues : unsigned long long {
    charTyp = 255, shortTyp = 65535, intTyp = 65535,
    longTyp = 4294967295UL,
    long_longTyp = 18446744073709551615ULL
};
```

If we do not specify the underlying type, then by default scoped enums have `int` as the underlying type. There is no default for unscoped enums; all we know is that the underlying type is large enough to hold the enumerator values. When the underlying type is specified (including implicitly specified for a scoped enum), it is an error for an enumerator to have a value that is too large to fit in that type.

Being able to specify the underlying type of an enum lets us control the type used across different implementations. We can be confident that our program compiled under one implementation will generate the same code when we compile it on another.

## Forward Declarations for Enumerations

**C++ 11** Under the new standard, we can forward declare an enum. An enum forward declaration must specify (implicitly or explicitly) the underlying size of the enum:

```
// forward declaration of unscoped enum named intValues
enum intValues : unsigned long long; // unscoped, must specify a type
enum class open_modes; // scoped enums can use int by default
```

Because there is no default size for an unscoped enum, every declaration must include the size of that enum. We can declare a scoped enum without specifying a size, in which case the size is implicitly defined as `int`.

As with any declaration, all the declarations and the definition of a given enum must match one another. In the case of enums, this requirement means that the size of the enum must be the same across all declarations and the enum definition. Moreover, we cannot declare a name as an unscoped enum in one context and redeclare it as a scoped enum later:

```
// error: declarations and definition must agree whether the enum is scoped or unscoped
enum class intValues;
enum intValues; // error: intValues previously declared as scoped enum
enum intValues : long; // error: intValues previously declared as int
```

## Parameter Matching and Enumerations

Because an object of enum type may be initialized only by another object of that enum type or by one of its enumerators (§ 19.3, p. 833), an integral value that happens to have the same value as an enumerator cannot be used to call a function expecting an enum argument:

```
// unscoped enumeration; the underlying type is machine dependent
enum Tokens { INLINE = 128, VIRTUAL = 129 };

void ff(Tokens);
void ff(int);

int main() {
    Tokens curTok = INLINE;
    ff(128);      // exactly matches ff(int)
    ff(INLINE);   // exactly matches ff(Tokens)
    ff(curTok);   // exactly matches ff(Tokens)
    return 0;
}
```

Although we cannot pass an integral value to an enum parameter, we can pass an object or enumerator of an unscoped enumeration to a parameter of integral type. When we do so, the enum value promotes to int or to a larger integral type. The actual promotion type depends on the underlying type of the enumeration:

```
void newf(unsigned char);
void newf(int);
unsigned char uc = VIRTUAL;
newf(VIRTUAL); // calls newf(int)
newf(uc);       // calls newf(unsigned char)
```

The enum Tokens has only two enumerators, the larger of which has the value 129. That value can be represented by the type unsigned char, and many compilers will use unsigned char as the underlying type for Tokens. Regardless of its underlying type, objects and the enumerators of Tokens are promoted to int. Enumerators and values of an enum type are not promoted to unsigned char, even if the values of the enumerators would fit.

## 19.4 Pointer to Class Member

A **pointer to member** is a pointer that can point to a nonstatic member of a class. Normally a pointer points to an object, but a pointer to member identifies a member of a class, not an object of that class. static class members are not part of any object, so no special syntax is needed to point to a static member. Pointers to static members are ordinary pointers.

The type of a pointer to member embodies both the type of a class and the type of a member of that class. We initialize such pointers to point to a specific member of a class without identifying an object to which that member belongs. When we use a pointer to member, we supply the object whose member we wish to use.

To explain pointers to members, we'll use a version of the `Screen` class from § 7.3.1 (p. 271):

```
class Screen {
public:
    typedef std::string::size_type pos;
    char get_cursor() const { return contents[cursor]; }
    char get() const;
    char get(pos ht, pos wd) const;
private:
    std::string contents;
    pos cursor;
    pos height, width;
};
```

### 19.4.1 Pointers to Data Members

As with any pointer, we declare a pointer to member using a `*` to indicate that the name we're declaring is a pointer. Unlike ordinary pointers, a pointer to member also incorporates the class that contains the member. Hence, we must precede the `*` with `classname::` to indicate that the pointer we are defining can point to a member of `classname`. For example:

```
// pdata can point to a string member of a const (or nonconst) Screen object
const string Screen::*pdata;
```

declares that `pdata` is a “pointer to a member of class `Screen` that has type `const string`.” The data members in a `const` object are themselves `const`. By making our pointer a pointer to `const string` member, we say that we can use `pdata` to point to a member of any `Screen` object, `const` or not. In exchange we can use `pdata` to read, but not write to, the member to which it points.

When we initialize (or assign to) a pointer to member, we say to which member it points. For example, we can make `pdata` point to the `contents` member of an unspecified `Screen` object as follows:

```
pdata = &Screen::contents;
```

Here, we apply the address-of operator not to an object in memory but to a member of the class `Screen`.

Of course, under the new standard, the easiest way to declare a pointer to member is to use `auto` or `decltype`:

```
auto pdata = &Screen::contents;
```

### Using a Pointer to Data Member

It is essential to understand that when we initialize or assign a pointer to member, that pointer does not yet point to any data. It identifies a specific member but not the object that contains that member. We supply the object when we dereference the pointer to member.

Analogous to the member access operators, `.` and `->`, there are two pointer-to-member access operators, `.*` and `->*`, that let us supply an object and dereference the pointer to fetch a member of that object:

```
Screen myScreen, *pScreen = &myScreen;
// .* dereferences pdata to fetch the contents member from the object myScreen
auto s = myScreen.*pdata;
// ->* dereferences pdata to fetch contents from the object to which pScreen points
s = pScreen->*pdata;
```

Conceptually, these operators perform two actions: They dereference the pointer to member to get the member that we want; then, like the member access operators, they fetch that member from an object (`.*`) or through a pointer (`->*`).

## A Function Returning a Pointer to Data Member

Normal access controls apply to pointers to members. For example, the `contents` member of `Screen` is `private`. As a result, the use of `pdata` above must have been inside a member or friend of class `Screen` or it would be an error.

Because data members are typically `private`, we normally can't get a pointer to data member directly. Instead, if a class like `Screen` wanted to allow access to its `contents` member, it would define a function to return a pointer to that member:

```
class Screen {
public:
    // data is a static member that returns a pointer to member
    static const std::string Screen::*data()
        { return &Screen::contents; }
    // other members as before
};
```

Here we've added a `static` member to class `Screen` that returns a pointer to the `contents` member of a `Screen`. The return type of this function is the same type as our original `pdata` pointer. Reading the return type from right to left, we see that `data` returns a pointer to a member of class `Screen` that is a `string` that is `const`. The body of the function applies the address-of operator to the `contents` member, so the function returns a pointer to the `contents` member of `Screen`.

When we call `data`, we get a pointer to member:

```
// data() returns a pointer to the contents member of class Screen
const string Screen::*pdata = Screen::data();
```

As before, `pdata` points to a member of class `Screen` but not to actual data. To use `pdata`, we must bind it to an object of type `Screen`:

```
// fetch the contents of the object named myScreen
auto s = myScreen.*pdata;
```

## EXERCISES SECTION 19.4.1

**Exercise 19.11:** What is the difference between an ordinary data pointer and a pointer to a data member?

**Exercise 19.12:** Define a pointer to member that can point to the `cursor` member of class `Screen`. Fetch the value of `Screen::cursor` through that pointer.

**Exercise 19.13:** Define the type that can represent a pointer to the `bookNo` member of the `Sales_data` class.

## 19.4.2 Pointers to Member Functions

We can also define a pointer that can point to a member function of a class. As with pointers to data members, the easiest way to form a pointer to member function is to use `auto` to deduce the type for us:

```
// pmf is a pointer that can point to a Screen member function that is const
// that returns a char and takes no arguments
auto pmf = &Screen::get_cursor;
```

Like a pointer to data member, a pointer to a function member is declared using `classname::*`. Like any other function pointer (§ 6.7, p. 247), a pointer to member function specifies the return type and parameter list of the type of function to which this pointer can point. If the member function is a `const` member (§ 7.1.2, p. 258) or a reference member (§ 13.6.3, p. 546), we must include the `const` or `reference` qualifier as well.

As with normal function pointers, if the member is overloaded, we must distinguish which function we want by declaring the type explicitly (§ 6.7, p. 248). For example, we can declare a pointer to the two-parameter version of `get` as

```
char (Screen::*pmf2)(Screen::pos, Screen::pos) const;
pmf2 = &Screen::get;
```

The parentheses around `Screen::*` in this declaration are essential due to precedence. Without the parentheses, the compiler treats the following as an (invalid) function declaration:

```
// error: nonmember function p cannot have a const qualifier
char Screen::*p(Screen::pos, Screen::pos) const;
```

This declaration tries to define an ordinary function named `p` that returns a pointer to a member of class `Screen` that has type `char`. Because it declares an ordinary function, the declaration can't be followed by a `const` qualifier.

Unlike ordinary function pointers, there is no automatic conversion between a member function and a pointer to that member:

```
// pmf points to a Screen member that takes no arguments and returns char
pmf = &Screen::get; // must explicitly use the address-of operator
pmf = Screen::get; // error: no conversion to pointer for member functions
```

## Using a Pointer to Member Function

As when we use a pointer to a data member, we use the `.*` or `->*` operators to call a member function through a pointer to member:

```
Screen myScreen, *pScreen = &myScreen;
// call the function to which pmf points on the object to which pScreen points
char c1 = (pScreen->*pmf)();
// passes the arguments 0, 0 to the two-parameter version of get on the object myScreen
char c2 = (myScreen.*pmf2)(0, 0);
```

The calls `(myScreen->*pmf)()` and `(pScreen.*pmf2)(0, 0)` require the parentheses because the precedence of the call operator is higher than the precedence of the pointer-to-member operators.

Without the parentheses,

```
myScreen.*pmf()
```

would be interpreted to mean

```
myScreen.(pmf())
```

This code says to call the function named `pmf` and use its return value as the operand of the pointer-to-member operator `(. *)`. However, `pmf` is not a function, so this code is in error.



Because of the relative precedence of the call operator, declarations of pointers to member functions and calls through such pointers must use parentheses: `(C::*p)(parms)` and `(obj.*p)(args)`.

## Using Type Aliases for Member Pointers

Type aliases or `typedefs` (§ 2.5.1, p. 67) make pointers to members considerably easier to read. For example, the following type alias defines `Action` as an alternative name for the type of the two-parameter version of `get`:

```
// Action is a type that can point to a member function of Screen
// that returns a char and takes two pos arguments
using Action =
char (Screen::*)(Screen::pos, Screen::pos) const;
```

`Action` is another name for the type “pointer to a `const` member function of class `Screen` taking two parameters of type `pos` and returning `char`.” Using this alias, we can simplify the definition of a pointer to `get` as follows:

```
Action get = &Screen::get; // get points to the get member of Screen
```

As with any other function pointer, we can use a pointer-to-member function type as the return type or as a parameter type in a function. Like any other parameter, a pointer-to-member parameter can have a default argument:

```
// action takes a reference to a Screen and a pointer to a Screen member function
Screen& action(Screen&, Action = &Screen::get);
```

`action` is a function taking two parameters, which are a reference to a `Screen` object and a pointer to a member function of class `Screen` that takes two pos parameters and returns a `char`. We can call `action` by passing it either a pointer or the address of an appropriate member function in `Screen`:

```
Screen myScreen;
// equivalent calls:
action(myScreen);           // uses the default argument
action(myScreen, get);      // uses the variable get that we previously defined
action(myScreen, &Screen::get); // passes the address explicitly
```



Type aliases make code that uses pointers to members much easier to read and write.

## Pointer-to-Member Function Tables

One common use for function pointers and for pointers to member functions is to store them in a function table (§ 14.8.3, p. 577). For a class that has several members of the same type, such a table can be used to select one from the set of these members. Let's assume that our `Screen` class is extended to contain several member functions, each of which moves the cursor in a particular direction:

```
class Screen {
public:
    // other interface and implementation members as before
    Screen& home();          // cursor movement functions
    Screen& forward();
    Screen& back();
    Screen& up();
    Screen& down();
};
```

Each of these new functions takes no parameters and returns a reference to the `Screen` on which it was invoked.

We might want to define a `move` function that can call any one of these functions and perform the indicated action. To support this new function, we'll add a static member to `Screen` that will be an array of pointers to the cursor movement functions:

```
class Screen {
public:
    // other interface and implementation members as before
    // Action is a pointer that can be assigned any of the cursor movement members
    using Action = Screen& (Screen::*)();
    // specify which direction to move; enum see § 19.3 (p. 832)
    enum Directions { HOME, FORWARD, BACK, UP, DOWN };
    Screen& move(Directions);
private:
    static Action Menu[];        // function table
};
```

The array named `Menu` will hold pointers to each of the cursor movement functions. Those functions will be stored at the offsets corresponding to the enumerators in `Directions`. The `move` function takes an enumerator and calls the appropriate function:

```
Screen& Screen::move(Directions cm)
{
    // run the element indexed by cm on this object
    return (this->*Menu[cm])(); // Menu[cm] points to a member function
}
```

The call inside `move` is evaluated as follows: The `Menu` element indexed by `cm` is fetched. That element is a pointer to a member function of the `Screen` class. We call the member function to which that element points on behalf of the object to which `this` points.

When we call `move`, we pass it an enumerator that indicates which direction to move the cursor:

```
Screen myScreen;
myScreen.move(Screen::HOME); // invokes myScreen.home
myScreen.move(Screen::DOWN); // invokes myScreen.down
```

What's left is to define and initialize the table itself:

```
Screen::Action Screen::Menu[] = { &Screen::home,
                                  &Screen::forward,
                                  &Screen::back,
                                  &Screen::up,
                                  &Screen::down,
};
```

### EXERCISES SECTION 19.4.2

**Exercise 19.14:** Is the following code legal? If so, what does it do? If not, why?

```
auto pmf = &Screen::get_cursor;
pmf = &Screen::get;
```

**Exercise 19.15:** What is the difference between an ordinary function pointer and a pointer to a member function?

**Exercise 19.16:** Write a type alias that is a synonym for a pointer that can point to the `avg_price` member of `Sales_data`.

**Exercise 19.17:** Define a type alias for each distinct `Screen` member function type.

### 19.4.3 Using Member Functions as Callable Objects

As we've seen, to make a call through a pointer to member function, we must use the `.*` or `->*` operators to bind the pointer to a specific object. As a result,

unlike ordinary function pointers, a pointer to member is *not* a callable object; these pointers do not support the function-call operator (§ 10.3.2, p. 388).

Because a pointer to member is not a callable object, we cannot directly pass a pointer to a member function to an algorithm. As an example, if we wanted to find the first empty string in a vector of strings, the obvious call won't work:

```
auto fp = &string::empty; // fp points to the string empty function
// error: must use .* or ->* to call a pointer to member
find_if(svec.begin(), svec.end(), fp);
```

The `find_if` algorithm expects a callable object, but we've supplied `fp`, which is a pointer to a member function. This call won't compile, because the code inside `find_if` executes a statement something like

```
// check whether the given predicate applied to the current element yields true
if (fp(*it)) // error: must use ->* to call through a pointer to member
```

which attempts to call the object it was passed.

## Using function to Generate a Callable

One way to obtain a callable from a pointer to member function is by using the library function template (§ 14.8.3, p. 577):

```
function<bool (const string&)> fcn = &string::empty;
find_if(svec.begin(), svec.end(), fcn);
```

Here we tell `function` that `empty` is a function that can be called with a `string` and returns a `bool`. Ordinarily, the object on which a member function executes is passed to the implicit `this` parameter. When we want to use `function` to generate a callable for a member function, we have to "translate" the code to make that implicit parameter explicit.

When a `function` object holds a pointer to a member function, the `function` class knows that it must use the appropriate pointer-to-member operator to make the call. That is, we can imagine that `find_if` will have code something like

```
// assuming it is the iterator inside find_if, so *it is an object in the given range
if (fcn(*it)) // assuming fcn is the name of the callable inside find_if
```

which `function` will execute using the proper pointer-to-member operator. In essence, the `function` class will transform this call into something like

```
// assuming it is the iterator inside find_if, so *it is an object in the given range
if (((*it).*p)()) // assuming p is the pointer to member function inside fcn
```

When we define a `function` object, we must specify the function type that is the signature of the callable objects that object can represent. When the callable is a member function, the signature's first parameter must represent the (normally implicit) object on which the member will be run. The signature we give to `function` must specify whether the object will be passed as a pointer or a reference.

When we defined `fcn`, we knew that we wanted to call `find_if` on a sequence of `string` objects. Hence, we asked `function` to generate a callable that took `string` objects. Had our `vector` held pointers to `string`, we would have told `function` to expect a pointer:

```

vector<string*> pvec;
function<bool (const string*)> fp = &string::empty;
// fp takes a pointer to string and uses the ->* to call empty
find_if(pvec.begin(), pvec.end(), fp);

```

## Using `mem_fn` to Generate a Callable

To use `function`, we must supply the call signature of the member we want to call. We can, instead, let the compiler deduce the member's type by using another library facility, `mem_fn`, which, like `function`, is defined in the functional header. Like `function`, `mem_fn` generates a callable object from a pointer to member. Unlike `function`, `mem_fn` will deduce the type of the callable from the type of the pointer to member:

```
find_if(svec.begin(), svec.end(), mem_fn(&string::empty));
```

Here we used `mem_fn(&string::empty)` to generate a callable object that takes a `string` argument and returns a `bool`.

The callable generated by `mem_fn` can be called on either an object or a pointer:

```

auto f = mem_fn(&string::empty); // f takes a string or a string*
f(*svec.begin()); // ok: passes a string object; f uses .* to call empty
f(&svec[0]); // ok: passes a pointer to string; f uses .-> to call empty

```

Effectively, we can think of `mem_fn` as if it generates a callable with an overloaded function call operator—one that takes a `string*` and the other a `string&`.

## Using `bind` to Generate a Callable

For completeness, we can also use `bind` (§ 10.3.4, p. 397) to generate a callable from a member function:

```

// bind each string in the range to the implicit first argument to empty
auto it = find_if(svec.begin(), svec.end(),
                  bind(&string::empty, _1));

```

As with `function`, when we use `bind`, we must make explicit the member function's normally implicit parameter that represents the object on which the member function will operate. Like `mem_fn`, the first argument to the callable generated by `bind` can be either a pointer or a reference to a `string`:

```

auto f = bind(&string::empty, _1);
f(*svec.begin()); // ok: argument is a string f will use .* to call empty
f(&svec[0]); // ok: argument is a pointer to string f will use .-> to call empty

```

C++  
11

## 19.5 Nested Classes

A class can be defined within another class. Such a class is a **nested class**, also referred to as a **nested type**. Nested classes are most often used to define implementation classes, such as the `QueryResult` class we used in our text query example (§ 12.3, p. 484).

### EXERCISES SECTION 19.4.3

**Exercise 19.18:** Write a function that uses `count_if` to count how many empty strings there are in a given vector.

**Exercise 19.19:** Write a function that takes a `vector<Sales_data>` and finds the first element whose average price is greater than some given amount.

Nested classes are independent classes and are largely unrelated to their enclosing class. In particular, objects of the enclosing and nested classes are independent from each other. An object of the nested type does not have members defined by the enclosing class. Similarly, an object of the enclosing class does not have members defined by the nested class.

The name of a nested class is visible within its enclosing class scope but not outside the class. Like any other nested name, the name of a nested class will not collide with the use of that name in another scope.

A nested class can have the same kinds of members as a nonnested class. Just like any other class, a nested class controls access to its own members using access specifiers. The enclosing class has no special access to the members of a nested class, and the nested class has no special access to members of its enclosing class.

A nested class defines a type member in its enclosing class. As with any other member, the enclosing class determines access to this type. A nested class defined in the `public` part of the enclosing class defines a type that may be used anywhere. A nested class defined in the `protected` section defines a type that is accessible only by the enclosing class, its friends, and its derived classes. A `private` nested class defines a type that is accessible only to the members and friends of the enclosing class.

### Declaring a Nested Class

The `TextQuery` class from § 12.3.2 (p. 487) defined a companion class named `QueryResult`. The `QueryResult` class is tightly coupled to our `TextQuery` class. It would make little sense to use `QueryResult` for any other purpose than to represent the results of a query operation on a `TextQuery` object. To reflect this tight coupling, we'll make `QueryResult` a member of `TextQuery`.

```
class TextQuery {
public:
    class QueryResult; // nested class to be defined later
    // other members as in § 12.3.2 (p. 487)
};
```

We need to make only one change to our original `TextQuery` class—we declare our intention to define `QueryResult` as a nested class. Because `QueryResult` is a type member (§ 7.4.1, p. 284), we must declare `QueryResult` before we use it. In particular, we must declare `QueryResult` before we use it as the return type for the `query` member. The remaining members of our original class are unchanged.

## Defining a Nested Class outside of the Enclosing Class

Inside `TextQuery` we declared `QueryResult` but did not define it. As with member functions, nested classes must be declared inside the class but can be defined either inside or outside the class.

When we define a nested class outside its enclosing class, we must qualify the name of the nested class by the name of its enclosing class:

```
// we're defining the QueryResult class that is a member of class TextQuery
class TextQuery::QueryResult {
    // in class scope, we don't have to qualify the name of the QueryResult parameters
    friend std::ostream&
        print(std::ostream&, const QueryResult&);

public:
    // no need to define QueryResult::line_no; a nested class can use a member
    // of its enclosing class without needing to qualify the member's name
    QueryResult(std::string,
                std::shared_ptr<std::set<line_no>>,
                std::shared_ptr<std::vector<std::string>>);

    // other members as in § 12.3.2 (p. 487)
};
```

The only change we made compared to our original class is that we no longer define a `line_no` member in `QueryResult`. The members of `QueryResult` can access that name directly from `TextQuery`, so there is no need to define it again.



Until the actual definition of a nested class that is defined outside the class body is seen, that class is an incomplete type (§ 7.3.3, p. 278).

## Defining the Members of a Nested Class

In this version, we did not define the `QueryResult` constructor inside the class body. To define the constructor, we must indicate that `QueryResult` is nested within the scope of `TextQuery`. We do so by qualifying the nested class name with the name of its enclosing class:

```
// defining the member named QueryResult for the class named QueryResult
// that is nested inside the class TextQuery
TextQuery::QueryResult::QueryResult(string s,
                                    shared_ptr<set<line_no>> p,
                                    shared_ptr<vector<string>> f):
    sought(s), lines(p), file(f) { }
```

Reading the name of the function from right to left, we see that we are defining the constructor for class `QueryResult`, which is nested in the scope of class `TextQuery`. The code itself just stores the given arguments in the data members and has no further work to do.

## Nested-Class static Member Definitions

If `QueryResult` had declared a `static` member, its definition would appear outside the scope of the `TextQuery`. For example, assuming `QueryResult` had a

`static` member, its definition would look something like

```
// defines an int static member of QueryResult
// which is a class nested inside TextQuery
int TextQuery::QueryResult::static_mem = 1024;
```

## Name Lookup in Nested Class Scope

Normal rules apply for name lookup (§ 7.4.1, p. 283) inside a nested class. Of course, because a nested class is a nested scope, the nested class has additional enclosing class scopes to search. This nesting of scopes explains why we didn't define `line_no` inside the nested version of `QueryResult`. Our original `QueryResult` class defined this member so that its own members could avoid having to write `TextQuery::line_no`. Having nested the definition of our results class inside `TextQuery`, we no longer need this `typedef`. The nested `QueryResult` class can access `line_no` without specifying that `line_no` is defined in `TextQuery`.

As we've seen, a nested class is a type member of its enclosing class. Members of the enclosing class can use the name of a nested class the same way it can use any other type member. Because `QueryResult` is nested inside `TextQuery`, the `query` member of `TextQuery` can refer to the name `QueryResult` directly:

```
// return type must indicate that QueryResult is now a nested class
TextQuery::QueryResult
TextQuery::query(const string &sought) const
{
    // we'll return a pointer to this set if we don't find sought
    static shared_ptr<set<line_no>> nodata(new set<line_no>);
    // use find and not a subscript to avoid adding words to wm!
    auto loc = wm.find(sought);
    if (loc == wm.end())
        return QueryResult(sought, nodata, file); // not found
    else
        return QueryResult(sought, loc->second, file);
}
```

As usual, the return type is not yet in the scope of the class (§ 7.4, p. 282), so we start by noting that our function returns a `TextQuery::QueryResult` value. However, inside the body of the function, we can refer to `QueryResult` directly, as we do in the `return` statements.

## The Nested and Enclosing Classes Are Independent

Although a nested class is defined in the scope of its enclosing class, it is important to understand that there is no connection between the objects of an enclosing class and objects of its nested classe(s). A nested-type object contains only the members defined inside the nested type. Similarly, an object of the enclosing class has only those members that are defined by the enclosing class. It does not contain the data members of any nested classes.

More concretely, the second `return` statement in `TextQuery::query`

```
return QueryResult(sought, loc->second, file);
```

uses data members of the `TextQuery` object on which `query` was run to initialize a `QueryResult` object. We have to use these members to construct the `QueryResult` object we return because a `QueryResult` object does not contain the members of its enclosing class.

## EXERCISES SECTION 19.5

**Exercise 19.20:** Nest your `QueryResult` class inside `TextQuery` and rerun the programs you wrote to use `TextQuery` in § 12.3.2 (p. 490).

## 19.6 union: A Space-Saving Class

A **union** is a special kind of class. A union may have multiple data members, but at any point in time, only one of the members may have a value. When a value is assigned to one member of the union, all other members become undefined. The amount of storage allocated for a union is at least as much as is needed to contain its largest data member. Like any class, a union defines a new type.

Some, but not all, class features apply equally to unions. A union cannot have a member that is a reference, but it can have members of most other types, including, under the new standard, class types that have constructors or destructors. A union can specify protection labels to make members `public`, `private`, or `protected`. By default, like `structs`, members of a union are `public`.

A union may define member functions, including constructors and destructors. However, a union may not inherit from another class, nor may a union be used as a base class. As a result, a union may not have virtual functions.

### Defining a union

unions offer a convenient way to represent a set of mutually exclusive values of different types. As an example, we might have a process that handles different kinds of numeric or character data. That process might define a union to hold these values:

```
// objects of type Token have a single member, which could be of any of the listed types
union Token {
    // members are public by default
    char    cval;
    int     ival;
    double  dval;
};
```

A union is defined starting with the keyword `union`, followed by an (optional) name for the union and a set of member declarations enclosed in curly braces. This code defines a union named `Token` that can hold a value that is either a `char`, an `int`, or a `double`.

## Using a union Type

The name of a union is a type name. Like the built-in types, by default unions are uninitialized. We can explicitly initialize a union in the same way that we can explicitly initialize aggregate classes (§ 7.5.5, p. 298) by enclosing the initializer in a pair of curly braces:

```
Token first_token = {'a'}; // initializes the cval member
Token last_token;         // uninitialized Token object
Token *pt = new Token;    // pointer to an uninitialized Token object
```

If an initializer is present, it is used to initialize the first member. Hence, the initialization of `first_token` gives a value to its `cval` member.

The members of an object of union type are accessed using the normal member access operators:

```
last_token.cval = 'z';
pt->ival = 42;
```

Assigning a value to a data member of a union object makes the other data members undefined. As a result, when we use a union, we must always know what type of value is currently stored in the union. Depending on the types of the members, retrieving or assigning to the value stored in the union through the wrong data member can lead to a crash or other incorrect program behavior.

## Anonymous unions

An **anonymous union** is an unnamed union that does not include any declarations between the close curly that ends its body and the semicolon that ends the union definition (§ 2.6.1, p. 73). When we define an anonymous union the compiler automatically creates an unnamed object of the newly defined union type:

```
union {                  // anonymous union
    char   cval;
    int    ival;
    double dval;
}; // defines an unnamed object, whose members we can access directly
cval = 'c'; // assigns a new value to the unnamed, anonymous union object
ival = 42;  // that object now holds the value 42
```

The members of an anonymous union are directly accessible in the scope where the anonymous union is defined.



An anonymous union cannot have `private` or `protected` members, nor can an anonymous union define member functions.

## unions with Members of Class Type

Under earlier versions of C++, unions could not have members of a class type that defined its own constructors or copy-control members. Under the new standard, this restriction is lifted. However, unions with members that define their

own constructors and /or copy-control members are more complicated to use than unions that have members of built-in type.

When a union has members of built-in type, we can use ordinary assignment to change the value that the union holds. Not so for unions that have members of nontrivial class types. When we switch the union's value to and from a member of class type, we must construct or destroy that member, respectively: When we switch the union to a member of class type, we must run a constructor for that member's type; when we switch from that member, we must run its destructor.

When a union has members of built-in type, the compiler will synthesize the memberwise versions of the default constructor or copy-control members. The same is not true for unions that have members of a class type that defines its own default constructor or one or more of the copy-control members. If a union member's type defines one of these members, the compiler synthesizes the corresponding member of the union as deleted (§ 13.1.6, p. 508).

For example, the `string` class defines all five copy-control members and the default constructor. If a union contains a `string` and does not define its own default constructor or one of the copy-control members, then the compiler will synthesize that missing member as deleted. If a class has a union member that has a deleted copy-control member, then that corresponding copy-control operation(s) of the class itself will be deleted as well.

## Using a Class to Manage union Members

Because of the complexities involved in constructing and destroying members of class type, unions with class-type members ordinarily are embedded inside another class. That way the class can manage the state transitions to and from the member of class type. As an example, we'll add a `string` member to our union. We'll define our union as an anonymous union and make it a member of a class named `Token`. The `Token` class will manage the union's members.

To keep track of what type of value the union holds, we usually define a separate object known as a **discriminant**. A discriminant lets us discriminate among the values that the union can hold. In order to keep the union and its discriminant in sync, we'll make the discriminant a member of `Token` as well. Our class will define a member of an enumeration type (§ 19.3, p. 832) to keep track of the state of its union member.

The only functions our class will define are the default constructor, the copy-control members, and a set of assignment operators that can assign a value of one of our union's types to the union member:

```
class Token {
public:
    // copy control needed because our class has a union with a string member
    // defining the move constructor and move-assignment operator is left as an exercise
    Token(): tok(INT), ival{0} { }
    Token(const Token& t): tok(t.tok) { copyUnion(t); }
    Token &operator=(const Token& t);
    // if the union holds a string, we must destroy it; see § 19.1.2 (p. 824)
    ~Token() { if (tok == STR) sval~string(); }
```

```

    // assignment operators to set the differing members of the union
    Token &operator=(const std::string&);
    Token &operator=(char);
    Token &operator=(int);
    Token &operator=(double);

private:
    enum {INT, CHAR, DBL, STR} tok; // discriminant
    union {                                // anonymous union
        char   cval;
        int    ival;
        double dval;
        std::string sval;
    }; // each Token object has an unnamed member of this unnamed union type
    // check the discriminant and copy the union member as appropriate
    void copyUnion(const Token&);

};

```

Our class defines a nested, unnamed, unscoped enumeration (§ 19.3, p. 832) that we use as the type for the member named `tok`. We defined `tok` following the close curly and before the semicolon that ends the definition of the `enum`, which defines `tok` to have this unnamed `enum` type (§ 2.6.1, p. 73).

We'll use `tok` as our discriminant. When the union holds an `int` value, `tok` will have the value `INT`; if the union has a `string`, `tok` will be `STR`; and so on.

The default constructor initializes the discriminant and the union member to hold an `int` value of 0.

Because our union has a member with a destructor, we must define our own destructor to (conditionally) destroy the `string` member. Unlike ordinary members of a class type, class members that are part of a union are not automatically destroyed. The destructor has no way to know which type the union holds, so it cannot know which member to destroy.

Our destructor checks whether the object being destroyed holds a `string`. If so, the destructor explicitly calls the `string` destructor (§ 19.1.2, p. 824) to free the memory used by that `string`. The destructor has no work to do if the union holds a member of any of the built-in types.

## Managing the Discriminant and Destroying the `string`

The assignment operators will set `tok` and assign the corresponding member of the union. Like the destructor, these members must conditionally destroy the `string` before assigning a new value to the union:

```

Token &Token::operator=(int i)
{
    if (tok == STR) sval.~string(); // if we have a string, free it
    ival = i;                      // assign to the appropriate member
    tok = INT;                     // update the discriminant
    return *this;
}

```

If the current value in the union is a `string`, we must destroy that `string` before assigning a new value to the union. We do so by calling the `string` destructor.

Once we've cleaned up the `string` member, we assign the given value to the member that corresponds to the parameter type of the operator. In this case, our parameter is an `int`, so we assign to `ival`. We update the discriminant and return.

The `double` and `char` assignment operators behave identically to the `int` version and are left as an exercise. The `string` version differs from the others because it must manage the transition to and from the `string` type:

```
Token &Token::operator=(const std::string &s)
{
    if (tok == STR) // if we already hold a string, just do an assignment
        sval = s;
    else
        new(&sval) string(s); // otherwise construct a string
    tok = STR;           // update the discriminant
    return *this;
}
```

In this case, if the `union` already holds a `string`, we can use the normal `string` assignment operator to give a new value to that `string`. Otherwise, there is no existing `string` object on which to invoke the `string` assignment operator. Instead, we must construct a `string` in the memory that holds the `union`. We do so using `placement new` (§ 19.1.2, p. 824) to construct a `string` at the location in which `sval` resides. We initialize that `string` as a copy of our `string` parameter. We next update the discriminant and return.

## Managing Union Members That Require Copy Control

Like the type-specific assignment operators, the copy constructor and assignment operators have to test the discriminant to know how to copy the given value. To do this common work, we'll define a member named `copyUnion`.

When we call `copyUnion` from the copy constructor, the `union` member will have been default-initialized, meaning that the first member of the `union` will have been initialized. Because our `string` is not the first member, we know that the `union` member doesn't hold a `string`. In the assignment operator, it is possible that the `union` already holds a `string`. We'll handle that case directly in the assignment operator. That way `copyUnion` can assume that if its parameter holds a `string`, `copyUnion` must construct its own `string`:

```
void Token::copyUnion(const Token &t)
{
    switch (t.tok) {
        case Token::INT: ival = t.ival; break;
        case Token::CHAR: cval = t.cval; break;
        case Token::DBL: dval = t.dval; break;
        // to copy a string, construct it using placement new; see (§ 19.1.2 (p. 824))
        case Token::STR: new(&sval) string(t.sval); break;
    }
}
```

This function uses a `switch` statement (§ 5.3.2, p. 178) to test the discriminant. For

the built-in types, we assign the value to the corresponding member; if the member we are copying is a `string`, we construct it.

The assignment operator must handle three possibilities for its `string` member: Both the left-hand and right-hand operands might be a `string`; neither operand might be a `string`; or one but not both operands might be a `string`:

```
Token &Token::operator=(const Token &t)
{
    // if this object holds a string and t doesn't, we have to free the old string
    if (tok == STR && t.tok != STR) sval~string();
    if (tok == STR && t.tok == STR)
        sval = t.sval; // no need to construct a new string
    else
        copyUnion(t); // will construct a string if t.tok is STR
    tok = t.tok;
    return *this;
}
```

If the union in the left-hand operand holds a `string`, but the union in the right-hand does not, then we have to first free the old `string` before assigning a new value to the union member. If both unions hold a `string`, we can use the normal `string` assignment operator to do the copy. Otherwise, we call `copyUnion` to do the assignment. Inside `copyUnion`, if the right-hand operand is a `string`, we'll construct a new `string` in the union member of the left-hand operand. If neither operand is a `string`, then ordinary assignment will suffice.

## EXERCISES SECTION 19.6

**Exercise 19.21:** Write your own version of the `Token` class.

**Exercise 19.22:** Add a member of type `Sales_data` to your `Token` class.

**Exercise 19.23:** Add a move constructor and move assignment to `Token`.

**Exercise 19.24:** Explain what happens if we assign a `Token` object to itself.

**Exercise 19.25:** Write assignment operators that take values of each type in the union.

## 19.7 Local Classes

A class can be defined inside a function body. Such a class is called a **local class**. A local class defines a type that is visible only in the scope in which it is defined. Unlike nested classes, the members of a local class are severely restricted.



All members, including functions, of a local class must be completely defined inside the class body. As a result, local classes are much less useful than nested classes.

In practice, the requirement that members be fully defined within the class limits the complexity of the member functions of a local class. Functions in local classes are rarely more than a few lines of code. Beyond that, the code becomes difficult for the reader to understand.

Similarly, a local class is not permitted to declare `static` data members, there being no way to define them.

## Local Classes May Not Use Variables from the Function's Scope

The names from the enclosing scope that a local class can access are limited. A local class can access only type names, `static` variables (§ 6.1.1, p. 205), and enumerators defined within the enclosing local scopes. A local class may not use the ordinary local variables of the function in which the class is defined:

```
int a, val;
void foo(int val)
{
    static int si;
    enum Loc { a = 1024, b };
    // Bar is local to foo
    struct Bar {
        Loc locVal; // ok: uses a local type name
        int barVal;
        void fooBar(Loc l = a) // ok: default argument is Loc::a
        {
            barVal = val; // error: val is local to foo
            barVal = ::val; // ok: uses a global object
            barVal = si; // ok: uses a static local object
            locVal = b; // ok: uses an enumerator
        }
    };
    // ...
}
```

## Normal Protection Rules Apply to Local Classes

The enclosing function has no special access privileges to the `private` members of the local class. Of course, the local class could make the enclosing function a friend. More typically, a local class defines its members as `public`. The portion of a program that can access a local class is very limited. A local class is already encapsulated within the scope of the function. Further encapsulation through information hiding is often overkill.

## Name Lookup within a Local Class

Name lookup within the body of a local class happens in the same manner as for other classes. Names used in the declarations of the members of the class must be in scope before the use of the name. Names used in the definition of a member can appear anywhere in the class. If a name is not found as a class member, then

the search continues in the enclosing scope and then out to the scope enclosing the function itself.

## Nested Local Classes

It is possible to nest a class inside a local class. In this case, the nested class definition can appear outside the local-class body. However, the nested class must be defined in the same scope as that in which the local class is defined.

```
void foo()
{
    class Bar {
public:
    // ...
    class Nested;      // declares class Nested
};

// definition of Nested
class Bar::Nested {
    // ...
};

}
```

As usual, when we define a member outside a class, we must indicate the scope of the name. Hence, we defined `Bar::Nested`, which says that `Nested` is a class defined in the scope of `Bar`.

A class nested in a local class is itself a local class, with all the attendant restrictions. All members of the nested class must be defined inside the body of the nested class itself.

## 19.8 Inherently Nonportable Features

To support low-level programming, C++ defines some features that are inherently **nonportable**. A nonportable feature is one that is machine specific. Programs that use nonportable features often require reprogramming when they are moved from one machine to another. The fact that the sizes of the arithmetic types vary across machines (§ 2.1.1, p. 32) is one such nonportable feature that we have already used.

In this section we'll cover two additional nonportable features that C++ inherits from C: bit-fields and the `volatile` qualifier. We'll also cover linkage directives, which is a nonportable feature that C++ adds to those that it inherits from C.

### 19.8.1 Bit-fields

A class can define a (nonstatic) data member as a **bit-field**. A bit-field holds a specified number of bits. Bit-fields are normally used when a program needs to pass binary data to another program or to a hardware device.



The memory layout of a bit-field is machine dependent.

A bit-field must have integral or enumeration type (§ 19.3, p. 832). Ordinarily, we use an unsigned type to hold a bit-field, because the behavior of a signed bit-field is implementation defined. We indicate that a member is a bit-field by following the member name with a colon and a constant expression specifying the number of bits:

```
typedef unsigned int Bit;

class File {
    Bit mode: 2;           // mode has 2 bits
    Bit modified: 1;       // modified has 1 bit
    Bit prot_owner: 3;     // prot_owner has 3 bits
    Bit prot_group: 3;     // prot_group has 3 bits
    Bit prot_world: 3;     // prot_world has 3 bits
    // operations and data members of File
public:
    // file modes specified as octal literals; see § 2.1.3 (p. 38)
    enum modes { READ = 01, WRITE = 02, EXECUTE = 03 } ;
    File &open(modes);
    void close();
    void write();
    bool isRead() const;
    void setWrite();
};


```

The mode bit-field has two bits, modified only one, and the other members each have three bits. Bit-fields defined in consecutive order within the class body are, if possible, packed within adjacent bits of the same integer, thereby providing for storage compaction. For example, in the preceding declaration, the five bit-fields will (probably) be stored in a single unsigned int. Whether and how the bits are packed into the integer is machine dependent.

The address-of operator (`&`) cannot be applied to a bit-field, so there can be no pointers referring to class bit-fields.



Ordinarily it is best to make a bit-field an unsigned type. The behavior of bit-fields stored in a signed type is implementation defined.

## Using Bit-fields

A bit-field is accessed in much the same way as the other data members of a class:

```
void File::write()
{
    modified = 1;
    // ...
}
void File::close()
{
    if (modified)
        // ... save contents
}
```

Bit-fields with more than one bit are usually manipulated using the built-in bitwise operators (§ 4.8, p. 152):

```
File &File::open(File::modes m)
{
    mode |= READ;      // set the READ bit by default
    // other processing
    if (m & WRITE) // if opening READ and WRITE
        // processing to open the file in read/write mode
    return *this;
}
```

Classes that define bit-field members also usually define a set of inline member functions to test and set the value of the bit-field:

```
inline bool File::isRead() const { return mode & READ; }
inline void File::setWrite() { mode |= WRITE; }
```

## 19.8.2 **volatile** Qualifier



The precise meaning of **volatile** is inherently machine dependent and can be understood only by reading the compiler documentation. Programs that use **volatile** usually must be changed when they are moved to new machines or compilers.

Programs that deal directly with hardware often have data elements whose value is controlled by processes outside the direct control of the program itself. For example, a program might contain a variable updated by the system clock. An object should be declared **volatile** when its value might be changed in ways outside the control or detection of the program. The **volatile** keyword is a directive to the compiler that it should not perform optimizations on such objects.

The **volatile** qualifier is used in much the same way as the **const** qualifier. It is an additional modifier to a type:

```
volatile int display_register; // int value that might change
volatile Task *curr_task; // curr_task points to a volatile object
volatile int iax[max_size]; // each element in iax is volatile
volatile Screen bitmapBuf; // each member of bitmapBuf is volatile
```

There is no interaction between the **const** and **volatile** type qualifiers. A type can be both **const** and **volatile**, in which case it has the properties of both.

In the same way that a class may define **const** member functions, it can also define member functions as **volatile**. Only **volatile** member functions may be called on **volatile** objects.

§ 2.4.2 (p. 62) described the interactions between the **const** qualifier and pointers. The same interactions exist between the **volatile** qualifier and pointers. We can declare pointers that are **volatile**, pointers to **volatile** objects, and pointers that are **volatile** that point to **volatile** objects:

```

volatile int v;      // v is a volatile int
int *volatile vip; // vip is a volatile pointer to int
volatile int *ivp;  // ivp is a pointer to volatile int
// vivp is a volatile pointer to volatile int
volatile int *volatile vivp;
int *ip = &v;    // error: must use a pointer to volatile
*ivp = &v;      // ok: ivp is a pointer to volatile
vivp = &v;      // ok: vivp is a volatile pointer to volatile

```

As with `const`, we may assign the address of a `volatile` object (or copy a pointer to a `volatile` type) only to a pointer to `volatile`. We may use a `volatile` object to initialize a reference only if the reference is `volatile`.

## Synthesized Copy Does Not Apply to `volatile` Objects

One important difference between the treatment of `const` and `volatile` is that the synthesized copy/move and assignment operators cannot be used to initialize or assign from a `volatile` object. The synthesized members take parameters that are references to (nonvolatile) `const`, and we cannot bind a nonvolatile reference to a `volatile` object.

If a class wants to allow `volatile` objects to be copied, moved, or assigned, it must define its own versions of the copy or move operation. As one example, we might write the parameters as `const volatile` references, in which case we can copy or assign from any kind of `Foo`:

```

class Foo {
public:
    Foo(const volatile Foo&); // copy from a volatile object
    // assign from a volatile object to a nonvolatile object
    Foo& operator=(volatile const Foo&);
    // assign from a volatile object to a volatile object
    Foo& operator=(volatile const Foo&) volatile;
    // remainder of class Foo
};

```

Although we can define copy and assignment for `volatile` objects, a deeper question is whether it makes any sense to copy a `volatile` object. The answer to that question depends intimately on the reason for using `volatile` in any particular program.

### 19.8.3 Linkage Directives: `extern "C"`

C++ programs sometimes need to call functions written in another programming language. Most often, that other language is C. Like any name, the name of a function written in another language must be declared. As with any function, that declaration must specify the return type and parameter list. The compiler checks calls to functions written in another language in the same way that it handles ordinary C++ functions. However, the compiler typically must generate different

code to call functions written in other languages. C++ uses **linkage directives** to indicate the language used for any non-C++ function.



Mixing C++ with code written in any other language, including C, requires access to a compiler for that language that is compatible with your C++ compiler.

## Declaring a Non-C++ Function

A linkage directive can have one of two forms: single or compound. Linkage directives may not appear inside a class or function definition. The same linkage directive must appear on every declaration of a function.

As an example, the following declarations shows how some of the C functions in the `cstring` header might be declared:

```
// illustrative linkage directives that might appear in the C++ header <cstring>
// single-statement linkage directive
extern "C" size_t strlen(const char *);

// compound-statement linkage directive
extern "C" {
    int strcmp(const char*, const char*);
    char *strcat(char*, const char*);
}
```

The first form of a linkage directive consists of the `extern` keyword followed by a string literal, followed by an “ordinary” function declaration.

The string literal indicates the language in which the function is written. A compiler is required to support linkage directives for C. A compiler may provide linkage specifications for other languages, for example, `extern "Ada"`, `extern "FORTRAN"`, and so on.

## Linkage Directives and Headers

We can give the same linkage to several functions at once by enclosing their declarations inside curly braces following the linkage directive. These braces serve to group the declarations to which the linkage directive applies. The braces are otherwise ignored, and the names of functions declared within the braces are visible as if the functions were declared outside the braces.

The multiple-declaration form can be applied to an entire header file. For example, the C++ `cstring` header might look like

```
// compound-statement linkage directive
extern "C" {
#include <string.h>      // C functions that manipulate C-style strings
}
```

When a `#include` directive is enclosed in the braces of a compound-linkage directive, all ordinary function declarations in the header file are assumed to be functions written in the language of the linkage directive. Linkage directives can be

nested, so if a header contains a function with its own linkage directive, the linkage of that function is unaffected.



The functions that C++ inherits from the C library are permitted to be defined as C functions but are not required to be C functions—it's up to each C++ implementation to decide whether to implement the C library functions in C or C++.

## Pointers to `extern "C"` Functions

The language in which a function is written is part of its type. Hence, every declaration of a function defined with a linkage directive must use the same linkage directive. Moreover, pointers to functions written in other languages must be declared with the same linkage directive as the function itself:

```
// pf points to a C function that returns void and takes an int
extern "C" void (*pf)(int);
```

When `pf` is used to call a function, the function call is compiled assuming that the call is to a C function.

A pointer to a C function does not have the same type as a pointer to a C++ function. A pointer to a C function cannot be initialized or be assigned to point to a C++ function (and vice versa). As with any other type mismatch, it is an error to try to assign two pointers with different linkage directives:

```
void (*pf1)(int);           // points to a C++ function
extern "C" void (*pf2)(int); // points to a C function
pf1 = pf2; // error: pf1 and pf2 have different types
```



Some C++ compilers may accept the preceding assignment as a language extension, even though, strictly speaking, it is illegal.

## Linkage Directives Apply to the Entire Declaration

When we use a linkage directive, it applies to the function and any function pointers used as the return type or as a parameter type:

```
// f1 is a C function; its parameter is a pointer to a C function
extern "C" void f1(void(*)(int));
```

This declaration says that `f1` is a C function that doesn't return a value. It has one parameter, which is a pointer to a function that returns nothing and takes a single `int` parameter. The linkage directive applies to the function pointer as well as to `f1`. When we call `f1`, we must pass it the name of a C function or a pointer to a C function.

Because a linkage directive applies to all the functions in a declaration, we must use a type alias (§ 2.5.1, p. 67) if we wish to pass a pointer to a C function to a C++ function:

```
// FC is a pointer to a C function
extern "C" typedef void FC(int);
// f2 is a C++ function with a parameter that is a pointer to a C function
void f2(FC *);
```

## Exporting Our C++ Functions to Other Languages

By using the linkage directive on a function definition, we can make a C++ function available to a program written in another language:

```
// the calc function can be called from C programs
extern "C" double calc(double dparm) { /* ... */ }
```

When the compiler generates code for this function, it will generate code appropriate to the indicated language.

It is worth noting that the parameter and return types in functions that are shared across languages are often constrained. For example, we can almost surely not write a function that passes objects of a (nontrivial) C++ class to a C program. The C program won't know about the constructors, destructors, or other class-specific operations.

### PREPROCESSOR SUPPORT FOR LINKING TO C

To allow the same source file to be compiled under either C or C++, the preprocessor defines `__cplusplus` (two underscores) when we compile C++. Using this variable, we can conditionally include code when we are compiling C++:

```
#ifdef __cplusplus
// ok: we're compiling C++
extern "C"
#endif
int strcmp(const char*, const char*);
```

## Overloaded Functions and Linkage Directives

The interaction between linkage directives and function overloading depends on the target language. If the language supports overloaded functions, then it is likely that a compiler that implements linkage directives for that language would also support overloading of these functions from C++.

The C language does not support function overloading, so it should not be a surprise that a C linkage directive can be specified for only one function in a set of overloaded functions:

```
// error: two extern "C" functions with the same name
extern "C" void print(const char*);
extern "C" void print(int);
```

If one function among a set of overloaded functions is a C function, the other functions must all be C++ functions:

```
class SmallInt { /* ... */ };
class BigNum { /* ... */ };

// the C function can be called from C and C++ programs
// the C++ functions overload that function and are callable from C++
extern "C" double calc(double);
extern SmallInt calc(const SmallInt&);
extern BigNum calc(const BigNum&);
```

The C version of `calc` can be called from C programs and from C++ programs. The additional functions are C++ functions with class parameters that can be called only from C++ programs. The order of the declarations is not significant.

### EXERCISES SECTION 19.8.3

**Exercise 19.26:** Explain these declarations and indicate whether they are legal:

```
extern "C" int compute(int *, int);
extern "C" double compute(double *, double);
```

## 8.2 THE STANDARD `string` CLASS

*I try to catch every sentence, every word you and I say, and quickly lock all these sentences and words away in my literary storehouse because they might come in handy.*

ANTON CHEKHOV, *The Seagull*

In Section 8.1, we introduced C strings. These C strings were simply arrays of characters terminated with the null character '\0'. In order to manipulate these C strings, you needed to worry about all the details of handling arrays. For example, when you want to add characters to a C string and there is not enough room in the array, you must create another array to hold this longer string of characters. In short, C strings require the programmer to keep track of all the low-level details of how the C strings are stored in memory. This is a lot of extra work and a source of programmer errors. The ANSI/ISO standard for C++ specified that C++ must also have a class `string` that allows the programmer to treat strings as a basic data type without needing to worry about implementation details. In this section we introduce you to this `string` type.

### Introduction to the Standard Class `string`

The class `string` is defined in the library whose name is also `<string>`, and the definitions are placed in the `std` namespace. So, in order to use the class `string`, your code must contain the following (or something more or less equivalent):

```
#include <string>
using namespace std;
```

+ operator does concatenation

The class `string` allows you to treat string values and string expressions very much like values of a simple type. You can use the = operator to assign a value to a string variable, and you can use the + sign to concatenate two strings. For example, suppose `s1`, `s2`, and `s3` are objects of type `string` and both `s1` and `s2` have string values. Then `s3` can be set equal to the concatenation of the string value in `s1` followed by the string value in `s2` as follows:

```
s3 = s1 + s2;
```

There is no danger of `s3` being too small for its new string value. If the sum of the lengths of `s1` and `s2` exceeds the capacity of `s3`, then more space is automatically allocated for `s3`.

As we noted earlier in this chapter, quoted strings are really C strings and so they are not literally of type `string`. However, C++ provides automatic type casting of quoted strings to values of type `string`. So, you can use quoted strings as if they were literal values of type `string`, and we (and most others) will often refer to quoted strings as if they were values of type `string`. For example,

```
s3 = "Hello Mom!";
```

sets the value of the string variable `s3` to a string object with the same characters as in the C string "Hello Mom!".

The class `string` has a default constructor that initializes a `string` object to the empty string. The class `string` also has a second constructor that takes one argument that is a standard C string and so can be a quoted string. This second constructor initializes the `string` object to a value that represents the same string as its C-string argument. For example,

```
string phrase;
string noun("ants");
```

The first line declares the string variable `phrase` and initializes it to the empty string. The second line declares `noun` to be of type `string` and initializes it to a string value equivalent to the C string "ants". Most programmers when talking loosely would say that "noun is initialized to "ants"," but there really is a type conversion here. The quoted string "ants" is a C string, not a value of type `string`. The variable `noun` receives a string value that has the same characters as "ants" in the same order as "ants", but the string value is not terminated with the null character '\0'. In fact, in theory at least, you do not know or care whether the string value of `noun` is even stored in an array, as opposed to some other data structure.

There is an alternate notation for declaring a string variable and invoking a constructor. The following two lines are exactly equivalent:

```
string noun("ants");
string noun = "ants";
```

These basic details about the class `string` are illustrated in Display 8.4. Note that, as illustrated there, you can output `string` values using the operator <<.

Consider the following line from Display 8.4:

```
phrase = "I love " + adjective + " " + noun + "!";
```

C++ must do a lot of work to allow you to concatenate strings in this simple and natural fashion. The string constant "I love" is not an object of type `string`. A string constant like "I love" is stored as a C string (in other words, as a null-terminated array of characters). When C++ sees "I love" as an argument to +, it finds the definition (or overloading) of + that applies to a value such as "I love". There are overloadings of the + operator that have a C string on the left and a string on the right, as well as the reverse of this positioning. There is even a version that has a C string on both sides of the + and produces a `string` object as the value returned. Of course, there is also the overloading you expect, with the type `string` for both operands.

Converting  
C-string constants  
to the type  
`string`

C++ did not really need to provide all those overloading cases for +. If these overloadings were not provided, C++ would look for a constructor that could perform a type conversion to convert the C string "I love" to a value for which + did apply. In this case, the constructor with the one C-string parameter would perform just such a conversion. However, the extra overloads are presumably more efficient.

The class `string` is often thought of as a modern replacement for C strings. However, in C++ you cannot easily avoid also using C strings when you program with the class `string`.

**DISPLAY 8.4 Program Using the Class string**

```

1 //Demonstrates the standard class string.
2 #include <iostream>
3 #include <string>
4 using namespace std;

5 int main( )
6 {
7     string phrase;           Initialized to the empty string
8     string adjective("fried"), noun("ants");
9     string wish = "Bon appetit!"; Two ways of initializing a string variable

10    phrase = "I love " + adjective + " " + noun + "!";
11    cout << phrase << endl
12        << wish << endl;
13    return 0;
14 }
```

**Sample Dialogue**

I love fried ants!  
Bon appetit!

**The Class string**

The class string can be used to represent values that are strings of characters. The class string provides more versatile string representation than the C strings discussed in Section 8.1.

The class string is defined in the library that is also named `<string>`, and its definition is placed in the `std` namespace. So, programs that use the class string should contain the following (or something more or less equivalent):

```
#include <string>
using namespace std;
```

The class string has a default constructor that initializes the string object to the empty string and a constructor that takes a C string as an argument and initializes the string object to a value that represents the string given as the argument. For example:

```
string s1, s2("Hello");
```

## I/O with the Class `string`

You can use the insertion operator `<<` and `cout` to output `string` objects just as you do for data of other types. This is illustrated in Display 8.4. Input with the class `string` is a bit more subtle.

The extraction operator `>>` and `cin` work the same for `string` objects as for other data, but remember that the extraction operator ignores initial whitespace and stops reading when it encounters more whitespace. This is as true for strings as it is for other data. For example, consider the following code:

```
string s1, s2;
cin >> s1;
cin >> s2;
```

If the user types in

```
May the hair on your toes grow long and curly!
```

then `s1` will receive the value "May" with any leading (or trailing) whitespace deleted. The variable `s2` receives the string "the". Using the extraction operator `>>` and `cin`, you can only read in words; you cannot read in a line or other string that contains a blank. Sometimes this is exactly what you want, but sometimes it is not at all what you want.

If you want your program to read an entire line of input into a variable of type `string`, you can use the function `getline`. The syntax for using `getline` with `string` objects is a bit different from what we described for C strings in Section 8.1. You do not use `cin.getline`; instead, you make `cin` the first argument to `getline`.<sup>2</sup> (Thus, this version of `getline` is not a member function.)

```
string line;
cout << "Enter a line of input:\n";
getline(cin, line);
cout << line << "END OF OUTPUT\n";
```

When embedded in a complete program, this code produces a dialogue like the following:

```
Enter some input:
Do bedo to you!
Do bedo to you!END OF OUTPUT
```

If there were leading or trailing blanks on the line, then they too would be part of the string value read by `getline`. This version of `getline` is in the

---

<sup>2</sup>This is a bit ironic, since the class `string` was designed using more modern object-oriented techniques, and the notation it uses for `getline` is the old fashioned, less object-oriented notation. This is an accident of history. This `getline` function was defined after the `iostream` library was already in use, so the designers had little choice but to make this `getline` a stand-alone function.

library `<string>`. You can use a stream object connected to a text file in place of `cin` to do input from a file using `getline`.

You cannot use `cin` and `>>` to read in a blank character. If you want to read one character at a time, you can use `cin.get`, which we discussed in Chapter 6. The function `cin.get` reads values of type `char`, not of type `string`, but it can be helpful when handling `string` input. Display 8.5 contains a program that illustrates both `getline` and `cin.get` used for `string` input. The significance of the function `newLine` is explained in the Pitfall subsection entitled Mixing `cin >>` variable and `getline`.

#### DISPLAY 8.5 Program Using the Class `string` (part 1 of 2)

```
1 //Demonstrates getline and cin.get.
2 #include <iostream>
3 #include <string>
4
5 void newLine( );
6
7 int main( )
8 {
9     using namespace std;
10
11    string firstName, lastName, recordName;
12    string motto = "Your records are our records.";
13
14    cout << "Enter your first and last name:\n";
15    cin >> firstName >> lastName;
16    newLine( );
17
18    recordName = lastName + ", " + firstName;
19    cout << "Your name in our records is: ";
20    cout << recordName << endl;
21
22    cout << "Our motto is\n"
23        << motto << endl;
24    cout << "Please suggest a better (one-line) motto:\n";
25    getline(cin, motto);
26    cout << "Our new motto will be:\n";
27    cout << motto << endl;
28
29 return 0;
30 }
```

```
26 //Uses iostream:
27 void newLine( )
28 {
29     using namespace std;
```

(continued)

**DISPLAY 8.5 Program Using the Class `string` (part 2 of 2)**

```
31     char nextChar;
32     do
33     {
34         cin.get(nextChar);
35     } while (nextChar != '\n');
36 }
```

**Sample Dialogue**

```
Enter your first and last name:  
B'Elanna Torres  
Your name in our records is: Torres, B'Elanna  
Our motto is  
Your records are our records.  
Please suggest a better (one-line) motto:  
Our records go where no records dared to go before.  
Our new motto will be:  
Our records go where no records dared to go before.
```

**I/O with `string` Objects**

You can use the insertion operator `<<` with `cout` to output `string` objects. You can input a `string` with the extraction operator `>>` and `cin`. When using `>>` for input, the code reads in a `string` delimited with whitespace. You can use the function `getline` to input an entire line of text into a `string` object.

**EXAMPLES**

```
string greeting("Hello"), response, nextWord;  
cout << greeting << endl;  
getline(cin, response);  
cin >> nextWord;
```

**SELF-TEST EXERCISES**

15. Consider the following code (and assume that it is embedded in a complete and correct program and then run):

```
string s1, s2;
```

```

cout << "Enter a line of input:\n";
cin >> s1 >> s2;
cout << s1 << "*" << s2 << "<END OF OUTPUT";

```

If the dialogue begins as follows, what will be the next line of output?

```

Enter a line of input:
A string is a joy forever!

```

16. Consider the following code (and assume that it is embedded in a complete and correct program and then run):

```

string s;
cout << "Enter a line of input:\n";
getline(cin, s);
cout << s << "<END OF OUTPUT";

```

If the dialogue begins as follows, what will be the next line of output?

```

Enter a line of input:
A string is a joy forever!

```

## PROGRAMMING TIP More Versions of `getline`

So far, we have described the following way of using `getline`:

```

string line;
cout << "Enter a line of input:\n";
getline(cin, line);

```

This version stops reading when it encounters the end-of-line marker '`\n`'. There is a version that allows you to specify a different character to use as a stopping signal. For example, the following will stop when the first question mark is encountered:

```

string line;
cout << "Enter some input:\n";
getline(cin, line, '?');

```

It makes sense to use `getline` as if it were a *void* function, but it actually returns a reference to its first argument, which is `cin` in the code above. Thus, the following will read a line of text into `s1` and a string of nonwhitespace characters into `s2`:

```

string s1, s2;
getline(cin, s1) >> s2;

```

The invocation `getline (cin, s1)` returns a reference to `cin`, so that after the invocation of `getline`, the next thing to happen is equivalent to

```
cin >> s2;
```

This kind of use of `getline` seems to have been designed for use in a C++ quiz show rather than to meet any actual programming need, but it can come in handy sometimes.

### PITFALL Mixing `cin >> variable;` and `getline`



VideoNote  
Example using `cin` and `getline` with the `string` class

Take care in mixing input using `cin >> variable;` with input using `getline`. For example, consider the following code:

```
int n;
string line;
cin >> n;
getline(cin, line);
```

#### getline for Objects of the Class `string`

The `getline` function for `string` objects has two versions:

```
istream& getline(istream& ins, string& strVar,
                  char delimiter);
```

and

```
istream& getline(istream& ins, string& strVar);
```

The first version of this function reads characters from the `istream` object given as the first argument (always `cin` in this chapter), inserting the characters into the `string` variable `strVar` until an instance of the delimiter character is encountered. The delimiter character is removed from the input and discarded. The second version uses '`\n`' for the default value of `delimiter`; otherwise, it works the same.

These `getline` functions return their first argument (always `cin` in this chapter), but they are usually used as if they were `void` functions.

When this code reads the following input, you might expect the value of `n` to be set to 42 and the value of `line` to be set to a `string` value representing "Hello hitchhiker.":

```
42
Hello hitchhiker.
```

However, while `n` is indeed set to the value of 42, `line` is set equal to the empty string. What happened?

Using `cin >> n` skips leading whitespace on the input, but leaves the rest of the line, in this case just '`\n`', for the next input. A statement like

```
cin >> n;
```

always leaves something on the line for a following `getline` to read (even if it is just the '`\n`'). In this case, the `getline` sees the '`\n`' and stops reading, so `getline` reads an empty string. If you find your program appearing to mysteriously ignore input data, see if you have mixed these two kinds of input. You may need to use either the `newLine` function from Display 8.5 or the function `ignore` from the library `iostream`. For example,

```
cin.ignore(1000, '\n');
```

With these arguments, a call to the `ignore` member function will read and discard the entire rest of the line up to and including the '`\n`' (or until it discards 1000 characters if it does not find the end of the line after 1000 characters).

There can be other baffling problems with programs that use `cin` with both `>>` and `getline`. Moreover, these problems can come and go as you move from one C++ compiler to another. When all else fails, or if you want to be certain of portability, you can resort to character-by-character input using `cin.get`.

These problems can occur with any of the versions of `getline` that we discuss in this chapter. ■

## String Processing with the Class `string`

The class `string` allows you to perform the same operations that you can perform with the C strings we discussed in Section 8.1 and more. You can access the characters in a `string` object in the same way that you access array elements, so `string` objects have all the advantages of arrays of characters plus a number of advantages that arrays do not have, such as automatically increasing their capacity. If `lastName` is the name of a `string` object, then `lastName[i]` gives access to the `i`th character in the string represented by `lastName`. This use of array square brackets is illustrated in Display 8.6.

Display 8.6 also illustrates the member function `length`. Every `string` object has a member function named `length` that takes no arguments and returns the length of the string represented by the `string` object. Thus, not only can a `string` object be used like an array but the `length` member function makes it behave like a partially filled array that automatically keeps track of how many positions are occupied.

When used with an object of the class `string`, the array square brackets do not check for illegal indexes. If you use an illegal index (that is, an index that is greater than or equal to the length of the string in the object), then the results are unpredictable but are bound to be bad. You may just get strange behavior without any error message that tells you that the problem is an illegal index value.

There is a member function named `at` that does check for illegal index values. This member function behaves basically the same as the square brackets, except for two points: You use function notation with `at`, so instead of

**DISPLAY 8.6 A `string` Object Can Behave Like an Array**

```
1 //Demonstrates using a string object as if it were an array.
2 #include <iostream>
3 #include <string>
4 using namespace std;
5 int main( )
6 {
7     string firstName, lastName;
8     cout << "Enter your first and last name:\n";
9     cin >> firstName >> lastName;
10    cout << "Your last name is spelled:\n";
11    int i;
12    for (i = 0; i <lastName.length( ); i++)
13    {
14        cout << lastName[i] << " ";
15        lastName[i] = '-';
16    }
17    cout << endl;
18    for (i = 0; i <lastName.length( ); i++)
19        cout << lastName[i] << " "; //Places a "-" under each letter.
20    cout << endl;
21    cout << "Good day " << firstName << endl;
22    return 0;
23 }
```

**Sample Dialogue**

Enter your first and last name:

John Crichton

Your last name is spelled:

C r i c h t o n

— — — — —

Good day John

`a[i]`, you use `a.at(i)`; and the `at` member function checks to see if `i` evaluates to an illegal index. If the value of `i` in `a.at(i)` is an illegal index, then you should get a run-time error message telling you what is wrong. In the following two example code fragments, the attempted access is out of range, yet the first of these probably will not produce an error message, although it will be accessing a nonexistent indexed variable:

```
string str("Mary");
cout << str[6] << endl;
```

The second example, however, will cause the program to terminate abnormally, so you at least know that something is wrong:

```
string str("Mary");
cout << str.at(6) << endl;
```

But be warned that some systems give very poor error messages when `str.at(i)` has an illegal index `i`.

You can change a single character in the string by assigning a *char* value to the indexed variable, such as `str[i]`. This may also be done with the member function `at`. For example, to change the third character in the `string` object `str` to 'X', you can use either of the following code fragments:

```
str.at(2) = 'X';
```

or

```
str[2] = 'X';
```

As in an ordinary array of characters, character positions for objects of type `string` are indexed starting with 0, so the third character in a `string` is in index position 2.

Display 8.7 gives a partial list of the member functions of the class `string`. In many ways, objects of the class `string` are better behaved than the C strings we introduced in Section 8.1. In particular, the `==` operator on objects of the `string` class returns a result that corresponds to our intuitive notion of strings being equal—namely, it returns *true* if the two strings contain the same characters in the same order, and returns *false* otherwise. Similarly, the comparison operators `<`, `>`, `<=`, `>=` compare string objects using lexicographic ordering. (Lexicographic ordering is alphabetic ordering using the order of symbols given in the ASCII character set in Appendix 3. If the strings consist of all letters and are both either all uppercase or all lowercase letters, then for this case lexicographic ordering is the same as everyday alphabetical ordering.)

#### DISPLAY 8.7 Member Functions of the Standard Class `string` (part 1 of 2)

| Example                            | Remarks                                                                                                                                                |
|------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Constructors</b>                |                                                                                                                                                        |
| <code>string str;</code>           | Default constructor creates empty string object <code>str</code> .                                                                                     |
| <code>string str("sample");</code> | Creates a string object with data "sample".                                                                                                            |
| <code>string str(aString);</code>  | Creates a string object <code>str</code> that is a copy of <code>aString</code> ; <code>aString</code> is an object of the class <code>string</code> . |

(continued)

**DISPLAY 8.7 Member Functions of the Standard Class `string` (part 2 of 2)****Accessors**

|                                           |                                                                                                                                                                  |
|-------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>str[i]</code>                       | Returns read/write reference to character in <code>str</code> at index <code>i</code> . Does not check for illegal index.                                        |
| <code>str.at(i)</code>                    | Returns read/write reference to character in <code>str</code> at index <code>i</code> . Same as <code>str[i]</code> , but this version checks for illegal index. |
| <code>str.substr(position, length)</code> | Returns the substring of the calling object starting at <code>position</code> and having <code>length</code> characters.                                         |
| <code>str.length( )</code>                | Returns the length of <code>str</code> .                                                                                                                         |

**Assignment/Modifiers**

|                                      |                                                                                                          |
|--------------------------------------|----------------------------------------------------------------------------------------------------------|
| <code>str1 = str2;</code>            | Initializes <code>str1</code> to <code>str2</code> 's data                                               |
| <code>str1 += str2;</code>           | Character data of <code>str2</code> is concatenated to the end of <code>str1</code> .                    |
| <code>str.empty( )</code>            | Returns true if <code>str</code> is an empty string; false otherwise.                                    |
| <code>str1 + str2</code>             | Returns a string that has <code>str2</code> 's data concatenated to the end of <code>str1</code> 's data |
| <code>str.insert(pos, str2);</code>  | Inserts <code>str2</code> into <code>str</code> beginning at position <code>pos</code> .                 |
| <code>str.erase(pos, length);</code> | Removes substring of size <code>length</code> , starting at position <code>pos</code> .                  |

**Comparison**

|                                                           |                                                              |
|-----------------------------------------------------------|--------------------------------------------------------------|
| <code>str1 == str2</code> <code>str1 != str2</code>       | Compare for equality or inequality; returns a Boolean value. |
| <code>str1 &lt; str2</code> <code>str1 &gt; str2</code>   | Four comparisons. All are lexicographical comparisons.       |
| <code>str1 &lt;= str2</code> <code>str1 &gt;= str2</code> |                                                              |

**Finds**

|                                                |                                                                                                                                                                                   |
|------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>str.find(str1)</code>                    | Returns index of the first occurrence of <code>str1</code> in <code>str</code> . If <code>str1</code> is not found, then the special value <code>string::npos</code> is returned. |
| <code>str.find(str1, pos)</code>               | Returns index of the first occurrence of string <code>str1</code> in <code>str</code> ; the search starts at position <code>pos</code> .                                          |
| <code>str.find_first_of(str1, pos)</code>      | Returns the index of the first instance in <code>str</code> of any character in <code>str1</code> , starting the search at position <code>pos</code> .                            |
| <code>str.find_first_not_of (str1, pos)</code> | Returns the index of the first instance in <code>str</code> of any character not in <code>str1</code> , starting the search at position <code>pos</code> .                        |

The first three lines of `removePunct` declare variables for use in the function. The `for` loop runs through the characters of the parameters one at a time and tries to find them in the `punct` string. To do this, a string that is the substring of `s`, of length 1 at each character position, is extracted. The position of this substring in the `punct` string is determined using the `find` member function. If this one-character string is not in the `punct` string, then the one-character string is concatenated to the `noPunct` string that is to be returned.

### **= and == Are Different for strings and C Strings**

The operators `=`, `==`, `!=`, `<`, `>`, `<=`, `>=`, when used with the standard C++ type `string`, produce results that correspond to our intuitive notion of how strings compare. They do not misbehave as they do with the C strings, as we discussed in Section 8.1

## **SELF-TEST EXERCISES**

17. Consider the following code:

```
string s1, s2("Hello");
cout << "Enter a line of input:\n";
cin >> s1;
if (s1 == s2)
    cout << "Equal\n";
else
    cout << "Not equal\n";
```

If the dialogue begins as follows, what will be the next line of output?

```
Enter a line of input:
Hello friend!
```

18. What is the output produced by the following code?

```
string s1, s2("Hello");
s1 = s2;
s2[0] = 'J';
cout << s1 << " " << s2;
```

## **Converting Between `string` Objects and C Strings**

You have already seen that C++ will perform an automatic type conversion to allow you to store a C string in a variable of type `string`. For example, the following will work fine:

```
char aCString[] = "This is my C string.";
string stringVariable;
stringVariable = aCString;
```

However, the following will produce a compiler error message:

```
aCString = stringVariable; //ILLEGAL
```

The following is also illegal:

```
strcpy(aCString, stringVariable); //ILLEGAL
```

`strcpy` cannot take a `string` object as its second argument, and there is no automatic conversion of `string` objects to C strings, which is the problem we cannot seem to get away from.

To obtain the C string corresponding to a `string` object, you must perform an explicit conversion. This can be done with the `string` member function `c_str( )`. The correct version of the copying we have been trying to do is the following:

```
strcpy(aCString, stringVariable.c_str()); //Legal;
```

Note that you need to use the `strcpy` function to do the copying. The member function `c_str( )` returns the C string corresponding to the `string` calling object. As we noted earlier in this chapter, the assignment operator does not work with C strings. So, just in case you thought the following might work, we should point out that it too is illegal.

```
aCString = stringVariable.c_str(); //ILLEGAL
```

## Converting Between Strings and Numbers

Prior to C++11 it was a bit complicated to convert between strings and numbers, but in C++11 it is simply a matter of calling a function. Use `stof`, `stod`, `stoi`, or `stol` to convert a string to a `float`, `double`, `int`, or `long`, respectively. Use `to_string` to convert a numeric type to a string. These functions are illustrated in the following example:

```
int i;
double d;
string s;
i = stoi("35"); // Converts the string "35" to an integer 35
d = stod("2.5"); // Converts the string "2.5" to the double 2.5
s = to_string(d*2); // Converts the double 5.0 to a string
                     "5.0000"
cout << i << " " << d << " " << s << endl;
```

The output is 35 2.5 5.0000

## 8.3 VECTORS

"Well, I'll eat it," said Alice, "and if it makes me grow larger, I can reach the key; and if it makes me grow smaller, I can creep under the door; so either way I'll get into the garden...."

LEWIS CARROLL, *Alice's Adventures in Wonderland*

Vectors can be thought of as arrays that can grow (and shrink) in length while your program is running. In C++, once your program creates an array, it cannot change the length of the array. Vectors serve the same purpose as arrays except that they can change length while the program is running. Vectors are part of a standard C++ library known as the STL (Standard Template Library), which we cover in more detail in Chapter 18.

You need not read the previous sections of this chapter before covering this section.

### Vector Basics

Like an array, a vector has a base type, and like an array, a vector stores a collection of values of its base type. However, the syntax for a vector type and a vector variable declaration are different from the syntax for arrays.

You declare a variable `v` for a vector with base type `int` as follows:

```
vector<int> v;
```

Declaring a vector variable

The notation `vector <Base_Type>` is a **template class**, which means you can plug in any type for `Base_Type` and that will produce a class for vectors with that base type. You can think of this as specifying the base type for a vector in the same sense as you specify a base type for an array. You can use any type, including class types, as the base type for a vector. The notation `vector <int>` is a class name, and so the previous declaration of `v` as a vector of type `vector <int>` includes a call to the default constructor for the class `vector <int>`, which creates a vector object that is empty (has no elements).

Vector elements are indexed starting with 0, the same as arrays. The array square brackets notation can be used to read or change these elements, just as with an array. For example, the following changes the value of the `i`th element of the vector `v` and then outputs that changed value. (`i` is an `int` variable.)

```
v[i] = 42;  
cout << "The answer is " << v[i];
```

There is, however, a restriction on this use of the square brackets notation with vectors that is unlike the same notation used with arrays. You can use `v[i]` to change the value of the `i`th element. However, you cannot initialize the `i`th element using `v[i]`; you can only change an element that has already been given some value. To add an element to an index position of a vector for the first time, you would normally use the member function `push_back`.

You add elements to a vector in order of positions, first at position 0, then position 1, then 2, and so forth. The member function `push_back` adds an element in the next available position. For example, the following gives initial values to elements 0, 1, and 2 of the vector `sample`:

```
vector<double> sample;
sample.push_back(0.0);
sample.push_back(1.1);
sample.push_back(2.2);
```

In C++11 we can initialize a vector the same way we initialize an array:

```
vector<double> sample = {0.0, 1.1, 2.2};
```

The number of elements in a vector is called the `size` of the vector. The member function `size` can be used to determine how many elements are in a vector. For example, after the previously shown code is executed, `sample.size()` returns 3. You can write out all the elements currently in the vector `sample` as follows:

```
for (int i = 0; i < sample.size(); i++)
    cout << sample[i] << endl;
```

The function `size` returns a value of type `unsigned int`, not a value of type `int`. (The type `unsigned int` allows only nonnegative integer values.) This returned value should be automatically converted to type `int` when it needs to be of type `int`, but some compilers may warn you that you are using an `unsigned int` where an `int` is required. If you want to be very safe, you can always apply a type cast to convert the returned `unsigned int` to an `int` or, in cases like this `for` loop, use a loop control variable of type `unsigned int` as follows:

```
for (unsigned int i = 0; i < sample.size(); i++)
    cout << sample[i] << endl;
```

Equivalently, we could use the ranged `for` loop:

```
for (auto i : sample)
    cout << i << endl;
```

A simple demonstration illustrating some basic vector techniques is given in Display 8.9.

There is a vector constructor that takes one integer argument and will initialize the number of positions given as the argument. For example, if you declare `v` as follows:

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

then the first ten elements are initialized to 0, and `v.size()` would return 10. You can then set the value of the `i`th element using `v[i]` for values of `i` equal to 0 through 9. In particular, the following could immediately follow the declaration:

```
for (unsigned int i = 0; i < 10; i++)
    v[i] = i;
```

**DISPLAY 8.9 Using a Vector**

```
1 #include <iostream>
2 #include <vector>
3 using namespace std;
4
5 int main( )
6 {
7     vector<int> v;
8     cout << "Enter a list of positive numbers.\n"
9         << "Place a negative number at the end.\n";
10
11     int next;
12     cin >> next;
13     while (next > 0)
14     {
15         v.push_back(next);
16         cout << next << " added. ";
17         cout << "v.size( ) = " << v.size( ) << endl;
18         cin >> next;
19     }
20
21     cout << "You entered:\n";
22     for (unsigned int i = 0; i < v.size( ); i++)
23         cout << v[i] << " ";
24     cout << endl;
25
26     return 0;
27 }
```

**Sample Dialogue**

```
Enter a list of positive numbers.  
Place a negative number at the end.
```

```
2 4 6 8 -1  
2 added. v.size( ) = 1  
4 added. v.size( ) = 2  
6 added. v.size( ) = 3  
8 added. v.size( ) = 4  
You entered:  
2 4 6 8
```

To set the  $i$ th element, for  $i$  greater than or equal to 10, you would use `push_back`.

When you use the constructor with an integer argument, vectors of numbers are initialized to the zero of the number type. If the vector base type is a class type, the default constructor is used for initialization.

The vector definition is given in the library `vector`, which places it in the `std` namespace. Thus, a file that uses vectors would include the following (or something similar):

```
#include <vector>
using namespace std;
```

### PITFALL Using Square Brackets Beyond the Vector Size

If `v` is a vector and `i` is greater than or equal to `v.size()`, then the element `v[i]` does not yet exist and needs to be created by using `push_back` to add elements up to and including position `i`. If you try to set `v[i]` for `i` greater than or equal to `v.size()`, as in

```
v[i] = n;
```

then you may or may not get an error message, but your program will undoubtedly misbehave at some point. ■

### Vectors

Vectors are used very much like arrays are used, but a vector does not have a fixed size. If it needs more capacity to store another element, its capacity is automatically increased. Vectors are defined in the library `<vector>`, which places them in the `std` namespace. Thus, a file that uses vectors would include the following (or something similar):

```
#include <vector>
using namespace std;
```

The vector class for a given `Base_Type` is written `vector <Base_Type>`. Two sample vector declarations are

```
vector<int> v; //default constructor
                //producing an empty vector.
vector<AClass> record(20); //vector constructor
                           //for AClass to initialize
                           //20 elements.
```

Elements are added to a vector using the member function `push_back`, as illustrated below:

```
v.push_back(42);
```

Once an element position has received its first element, either with `push_back` or with a constructor initialization, that element position can then be accessed using square bracket notation, just like an array element.

## ■ PROGRAMMING TIP Vector Assignment Is Well Behaved

The assignment operator with vectors does an element-by-element assignment to the vector on the left-hand side of the assignment operator (increasing capacity if needed and resetting the size of the vector on the left-hand side of the assignment operator). Thus, provided the assignment operator on the base type makes an independent copy of the element of the base type, then the assignment operator on the vector will make an independent copy.

Note that for the assignment operator to produce a totally independent copy of the vector on the right-hand side of the assignment operator requires that the assignment operator on the base type make completely independent copies. The assignment operator on a vector is only as good (or bad) as the assignment operator on its base type. (Details on overloading the assignment operator for classes that need it are given in Chapter 11.) ■

## Efficiency Issues

At any point in time a vector has a **capacity**, which is the number of elements for which it currently has memory allocated. The member function `capacity()` can be used to find out the capacity of a vector. Do not confuse the capacity of a vector with the size of a vector. The `size` is the number of elements in a vector, while the `capacity` is the number of elements for which there is memory allocated. Typically, the capacity is larger than the size, and the capacity is always greater than or equal to the size.

Whenever a vector runs out of capacity and needs room for an additional member, the capacity is automatically increased. The exact amount of the increase is implementation-dependent but always allows for more capacity than is immediately needed. A commonly used implementation scheme is for the capacity to double whenever it needs to increase. Since increasing capacity is a complex task, this approach of reallocating capacity in large chunks is more efficient than allocating numerous small chunks.

### Size and Capacity

The `size` of a vector is the number of elements in the vector. The **capacity** of a vector is the number of elements for which it currently has memory allocated. For a vector `v`, the `size` and `capacity` can be recovered with the member functions `v.size()` and `v.capacity()`.

You can completely ignore the capacity of a vector and that will have no effect on what your program does. However, if efficiency is an issue, you might want to manage capacity yourself and not simply accept the default behavior of doubling capacity whenever more is needed. You can use the member function `reserve` to explicitly increase the capacity of a vector. For example,

```
v.reserve(32);
```

sets the capacity to at least 32 elements, and

```
v.reserve(v.size( ) + 10);
```

sets the capacity to at least 10 more than the number of elements currently in the vector. Note that you can rely on `v.reserve` to increase the capacity of a vector, but it does not necessarily decrease the capacity of a vector if the argument is smaller than the current capacity.

You can change the size of a vector using the member function `resize`. For example, the following resizes a vector to 24 elements:

```
v.resize(24);
```

If the previous size was less than 24, then the new elements are initialized as we described for the constructor with an integer argument. If the previous size was greater than 24, then all but the first 24 elements are lost. The capacity is automatically increased if need be. Using `resize` and `reserve`, you can shrink the size and capacity of a vector when there is no longer any need for some elements or some capacity.

## SELF-TEST EXERCISES

19. Is the following program legal? If so, what is the output?

```
#include <iostream>
#include <vector>
using namespace std;

int main( )
{
    vector<int> v(10);
    int i;

    for (i = 0; i < v.size( ); i++)
        v[i] = i;

    vector<int> copy;
    copy = v;
    v[0] = 42;

    for (i = 0; i < copy.size( ); i++)
        cout << copy[i] << " ";
    cout << endl;

    return 0;
}
```

20. What is the difference between the size and the capacity of a vector?

## CHAPTER SUMMARY

- A C-string variable is the same thing as an array of characters, but it is used in a slightly different way. A string variable uses the null character '\0' to mark the end of the string stored in the array.
- C-string variables usually must be treated like arrays, rather than simple variables of the kind we used for numbers and single characters. In particular, you cannot assign a C-string value to a C-string variable using the equal sign, =, and you cannot compare the values in two C-string variables using the == operator. Instead, you must use special C-string functions to perform these tasks.
- The ANSI/ISO standard <string> library provides a fully featured class called `string` that can be used to represent strings of characters.
- Objects of the class `string` are better behaved than C strings. In particular, the assignment and equal operators, = and ==, have their intuitive meaning when used with objects of the class `string`.
- Vectors can be thought of as arrays that can grow (and shrink) in length while your program is running.

### Answers to Self-Test Exercises

1. The following two are equivalent to each other (but not equivalent to any others):

```
char stringVar[10] = "Hello";
char stringVar[10] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

The following two are equivalent to each other (but not equivalent to any others):

```
char stringVar[6] = "Hello";
char stringVar[] = "Hello";
```

The following is not equivalent to any of the others:

```
char stringVar[10] = {'H', 'e', 'l', 'l', 'o'};
```

2. "DoBeDo to you"
3. The declaration means that `stringVar` has room for only six characters (including the null character '\0'). The function `strcat` does not check that there is room to add more characters to `stringVar`, so `strcat` will write all the characters in the string "and Good-bye." into memory, even though that requires more memory than has been assigned to `stringVar`. This means memory that should not be changed will be changed. The net effect is unpredictable, but bad.

- If the value to be inserted is greater than the current node's value:
  - then, if the current node's right pointer is `null`, set the current node's right pointer to point to the new node
  - If the current node's right pointer is not `null`, set the current node's pointer to the address of the right node and repeat from step 2.

Write a method in your base `BinaryTree` class named `printInOrder` which is recursive and operates as follows: for each `Node`, you should first follow the left pointer if it is not null, then print the current `Node`'s value and then follow the right pointer if it is not null.

Write a driver program which constructs a `BinaryTree` and a `BinarySearchTree` and insert the same values into both. Then call the `printInOrder` method on both trees. The `BinarySearchTree` object should print out in sorted order.



# Exception Handling 16

## 16.1 EXCEPTION-HANDLING BASICS 929

- A Toy Example of Exception Handling 929
- Defining Your Own Exception Classes 938
- Multiple Throws and Catches 938
- Pitfall:* Catch the More Specific Exception First 942
- Programming Tip:* Exception Classes Can Be Trivial 943
- Throwing an Exception in a Function 943
- Exception Specification 945
- Pitfall:* Exception Specification in Derived Classes 947

## 16.2 PROGRAMMING TECHNIQUES FOR EXCEPTION HANDLING 948

- When to Throw an Exception 948
- Pitfall:* Uncaught Exceptions 950
- Pitfall:* Nested `try-catch` Blocks 950
- Pitfall:* Overuse of Exceptions 950
- Exception Class Hierarchies 951
- Testing for Available Memory 951
- Rethrowing an Exception 952

*It's the exception that proves the rule.*

COMMON MAXIM (*possibly a corruption of something like: It's the exception that tests the rule.*)

## INTRODUCTION

One way to write a program is to first assume that nothing unusual or incorrect will happen. For example, if the program takes an entry off a list, you might assume that the list is not empty. Once you have the program working for the core situation where things always go as planned, you can then add code to take care of the exceptional cases. In C++, there is a way to reflect this approach in your code. Basically, you write your code as if nothing very unusual happens. After that, you use the C++ exception-handling facilities to add code for those unusual cases. Exception handling is commonly used to handle error situations, but perhaps a better way to view exceptions is as a way to handle “exceptional situations.” After all, if your code correctly handles an “error,” then it no longer is an error.

Perhaps the most important use of exceptions is to deal with functions that have some special case that is handled differently depending on how the function is used. Perhaps the function will be used in many programs, some of which will handle the special case in one way and some of which will handle it in some other way. For example, if there is a division by zero in the function, then it may turn out that for some invocations of the function, the program should end, but for other invocations of the function something else should happen. You will see that such a function can be defined to throw an exception if the special case occurs, and that exception will allow the special case to be handled outside of the function. That way, the special case can be handled differently for different invocations of the function.

In C++, exception handling proceeds as follows: Either some library software or your code provides a mechanism that signals when something unusual happens. This is called *throwing an exception*. At another place in your program, you place the code that deals with the exceptional case. This is called *handling the exception*. This method of programming makes for cleaner code. Of course, we still need to explain the details of how you do this in C++.

## PREREQUISITES

With the exception of one subsection that can be skipped, Section 16.1 uses material only from Chapters 2 to 6 and 10 to 11. The Pitfall subsection of Section 16.1 entitled “Exception Specification in Derived Classes” uses material from Chapter 15. This Pitfall subsection can be skipped without loss of continuity.

With the exception of one subsection that can be skipped, Section 16.2 uses material only from Chapters 2 to 8 and 10 to 12 and Section 15.1 of Chapter 15 in addition to Section 16.1. The subsection of Section 16.2 entitled “Testing for Available Memory” uses material from Chapter 15. This subsection can be skipped without loss of continuity.

## 16.1 EXCEPTION-HANDLING BASICS

*Well, the program works for most cases. I didn't know it had to work for that case.*

COMPUTER SCIENCE STUDENT, APPEALING A GRADE

Exception handling is meant to be used sparingly and in situations that are more involved than what is reasonable to include in a simple introductory example. So, we will teach you the exception-handling details of C++ by means of simple examples that would not normally use exception handling. This makes a lot of sense for learning about exception handling, but do not forget that these first examples are toy examples, and in practice, you would not use exception handling for anything that simple.

### A Toy Example of Exception Handling

For this example, suppose that milk is such an important food in our culture that people almost never run out of it, but still we would like our programs to accommodate the very unlikely situation of running out of milk. The basic code, which assumes we do not run out of milk, might be as follows:

```
cout << "Enter number of donuts:\n";
cin >> donuts;
cout << "Enter number of glasses of milk:\n";
cin >> milk;
dpg = donuts/static_cast<double>(milk);
cout << donuts << " donuts.\n"
    << milk << " glasses of milk.\n"
    << "You have " << dpg
    << " donuts for each glass of milk.\n";
```

If there is no milk, then this code will include a division by zero, which is an error. To take care of the special situation in which we run out of milk, we can add a test for this unusual situation. The complete program with this added test for the special situation is shown in Display 16.1. The program in Display 16.1 does not use exception handling. Now, let's see how this program can be rewritten using the C++ exception-handling facilities.

**DISPLAY 16.1 Handling a Special Case Without Exception Handling**

```
1  include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      int donuts, milk;
7      double dpg;
8      cout << "Enter number of donuts:\n";
9      cin >> donuts;
10     cout << "Enter number of glasses of milk:\n";
11     cin >> milk;
12
13     if (milk <= 0)
14     {
15         cout << donuts << " donuts, and No Milk!\n"
16             << "Go buy some milk.\n";
17     }
18     else
19     {
20         dpg = donuts/static_cast<double>(milk);
21         cout << donuts << " donuts.\n"
22             << milk << " glasses of milk.\n"
23             << "You have " << dpg
24             << " donuts for each glass of milk.\n";
25     }
26
27     cout << "End of program.\n";
28     return 0;
29 }
```

**Sample Dialogue**

```
Enter number of donuts:
12
Enter number of glasses of milk:
0
12 donuts, and No Milk!
Go buy some milk.
End of program.
```

In Display 16.2, we have rewritten the program from Display 16.1 using an exception. This is only a toy example, and you would probably not use an exception in this case. However, it does give us a simple example. Although the program as a whole is not simpler, at least the part between the words *try* and *catch* is cleaner, and this hints at the advantage of using exceptions. Look

**DISPLAY 16.2 Same Thing Using Exception Handling (part 1 of 2)**

```
1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     int donuts, milk;
7     double dpg;
8
9     try
10    {
11         cout << "Enter number of donuts:\n";
12         cin >> donuts;
13         cout << "Enter number of glasses of milk:\n";
14         cin >> milk;
15
16         if (milk <= 0)
17             throw donuts;
18
19         dpg = donuts/static_cast<double>(milk);
20         cout << donuts << " donuts.\n"
21             << milk << " glasses of milk.\n"
22             << "You have " << dpg
23             << " donuts for each glass of milk.\n";
24     }
25     catch(int e)
26     {
27         cout << e << " donuts, and No Milk!\n"
28             << "Go buy some milk.\n";
29     }
30
31     cout << "End of program.\n";
32     return 0;
33 }
```

**Sample Dialogue 1**

```
Enter number of donuts:
12
Enter number of glasses of milk:
6
12 donuts.
6 glasses of milk.
You have 2 donuts for each glass of milk.
```

*(continued)*

**DISPLAY 16.2 Same Thing Using Exception Handling (part 2 of 2)****Sample Dialogue 2**

```
Enter number of donuts:
```

```
12
```

```
Enter number of glasses of milk:
```

```
0
```

```
12 donuts, and No Milk!
```

```
Go buy some milk.
```

```
End of program.
```

at the code between the words *try* and *catch*. That code is basically the same as the code in Display 16.1, but rather than the big *if-else* statement (shown in color in Display 16.1) this new program has the following smaller *if* statement (plus some simple nonbranching statements):

```
if (milk <= 0)
    throw donuts;
```

This *if* statement says that if there is no milk, then do something exceptional. That something exceptional is given after the word *catch*. The idea is that the normal situation is handled by the code following the word *try*, and that the code following the word *catch* is used only in exceptional circumstances. We have thus separated the normal case from the exceptional case. In this toy example, this separation does not really buy us too much, but in other situations it will prove to be very helpful. Let's look at the details.

The basic way of handling exceptions in C++ consists of the *try-throw-catch* threesome. A *try* block has the syntax

```
try
{
    Some_Code
}
```

This *try* block contains the code for the basic algorithm that tells the computer what to do when everything goes smoothly. It is called a *try* block because you are not 100 percent sure that all will go smoothly, but you want to "give it a try."

Now if something *does* go wrong, you want to throw an exception, which is a way of indicating that something went wrong. The basic outline, when we add a *throw*, is as follows:

```
try
{
```

```
Code_To_Try  
Possibly_Throw_An_Exception  
More_Code  
}
```

The following is an example of a *try* block with a *throw* statement included (copied from Display 16.2):

```
try  
{  
    cout << "Enter number of donuts:\n";  
    cin >> donuts;  
    cout << "Enter number of glasses of milk:\n";  
    cin >> milk;  
    if (milk <= 0)  
        throw donuts;  
    dpg = donuts/static_cast<double>(milk);  
    cout << donuts << " donuts.\n"  
        << milk << " glasses of milk.\n"  
        << "You have " << dpg  
        << " donuts for each glass of milk.\n";  
}
```

The following statement **throws** the *int* value *donuts*:

```
throw donuts;
```

The value thrown, in this case *donuts*, is sometimes called an **exception**, and the execution of a *throw* statement is called **throwing an exception**. You can throw a value of any type. In this case, an *int* value is thrown.

## throw Statement

### SYNTAX

```
throw Expression_for_Value_to_Be_Thrown;
```

When the *throw* statement is executed, the execution of the enclosing *try* block is stopped. If the *try* block is followed by a suitable *catch* block, then flow of control is transferred to the *catch* block. A *throw* statement is almost always embedded in a branching statement, such as an *if* statement. The value thrown can be of any type.

### EXAMPLE

```
if (milk <= 0)  
    throw donuts;
```

As the name suggests, when something is “thrown,” something goes from one place to another place. In C++, what goes from one place to another is the flow of control (as well as the value thrown). When an exception is thrown, the code in the surrounding *try* block stops executing and another portion of code, known as a *catch block*, begins execution. This executing of the *catch* block is called catching the exception or handling the exception. When an exception is thrown, it should ultimately be handled by (caught by) some *catch* block. In Display 16.2, the appropriate *catch* block immediately follows the *try* block. We repeat the *catch* block here:

```
catch(int e)
{
    cout << e << " donuts, and No Milk!\n"
        << "Go buy some milk.\n";
}
```

This *catch* block looks very much like a function definition that has a parameter of a type *int*. It is not a function definition, but in some ways, a *catch* block is like a function. It is a separate piece of code that is executed when your program encounters (and executes) the following (within the preceding *try* block):

```
throw Some_int;
```

So, this *throw* statement is similar to a function call, but instead of calling a function, it calls the *catch* block and says to execute the code in the *catch* block. A *catch* block is often referred to as an **exception handler**, which is a term that suggests that a *catch* block has a function-like nature.

What is that identifier *e* in the following line from a *catch* block?

```
catch(int e)
```

That identifier *e* looks like a parameter and acts very much like a parameter. So, we will call this *e* the **catch-block parameter**. (But remember, this does not mean that the *catch* block is a function.) The *catch-block parameter* does two things:

1. The *catch-block parameter* is preceded by a type name that specifies what kind of thrown value the *catch* block can catch.
2. The *catch-block parameter* gives you a name for the thrown value that is caught, so you can write code in the *catch* block that does things with the thrown value that is caught.

We will discuss these two functions of the *catch-block parameter* in reverse order. In this subsection, we will discuss using the *catch-block parameter* as a name for the value that was thrown and is caught. In the subsection entitled “Multiple Throws and Catches,” later in this chapter, we will discuss which *catch* block (which exception handler) will process a value that is thrown. Our current example has only one *catch* block. A common

name for a *catch*-block parameter is *e*, but you can use any legal identifier in place of *e*.

Let's see how the *catch* block in Display 16.2 works. When a value is thrown, execution of the code in the *try* block ends and control passes to the *catch* block (or blocks) that are placed right after the *try* block. The *catch* block from Display 16.2 is reproduced here:

```
catch(int e)
{
    cout << e << " donuts, and No Milk!\n"
        << "Go buy some milk.\n";
}
```

When a value is thrown, the thrown value must be of type *int* in order for this particular *catch* block to apply. In Display 16.2, the value thrown is given by the variable *donuts*, and since *donuts* is of type *int*, this *catch* block can catch the value thrown.

Suppose the value of *donuts* is 12 and the value of *milk* is 0, as in the second sample dialogue in Display 16.2. Since the value of *milk* is not positive, the *throw* statement within the *if* statement is executed. In that case, the value of the variable *donuts* is thrown. When the *catch* block in Display 16.2 catches the value of *donuts*, the value of *donuts* is plugged in for the *catch*-block parameter *e* and the code in the *catch* block is executed, producing the following output:

```
12 donuts, and No Milk!
Go buy some milk.
```

If the value of *donuts* is positive, the *throw* statement is not executed. In this case, the entire *try* block is executed. After the last statement in the *try* block is executed, the statement after the *catch* block is executed. Note that if no exception is thrown, then the *catch* block is ignored.

This makes it sound like a *try-throw-catch* setup is equivalent to an *if-else* statement. It almost is equivalent, except for the value thrown. A *try-throw-catch* setup is similar to an *if-else* statement *with the added ability to send a message to one of the branches*. This does not sound much different from an *if-else* statement, but it turns out to be a big difference in practice.

To summarize in a more formal tone, a *try* block contains some code that we are assuming includes a *throw* statement. The *throw* statement is normally executed only in exceptional circumstances, but when it is executed, it throws a value of some type. When an exception (a value like *donuts* in Display 16.2) is thrown, that is the end of the *try* block. All the rest of the code in the *try* block is ignored and control passes to a suitable *catch* block. A *catch* block applies only to an immediately preceding *try* block. If the exception is thrown, then that exception object is plugged in for the *catch*-block parameter, and the statements in the *catch* block are executed. For example, if you look at the dialogues in Display 16.2, you will see that as soon

### catch-Block Parameter

The *catch*-block parameter is an identifier in the heading of a *catch* block that serves as a placeholder for an exception (a value) that might be thrown. When a (suitable) value is thrown in the preceding *try* block, that value is plugged in for the *catch*-block parameter. You can use any legal (nonreserved word) identifier for a *catch*-block parameter.

#### EXAMPLE

```
catch(int e)
{
    cout << e << " donuts, and No Milk!\n"
        << "Go buy some milk.\n";
}
```

e is the *catch*-block parameter.

as the user enters a nonpositive number, the *try* block stops and the *catch* block is executed. For now, we will assume that every *try* block is followed by an appropriate *catch* block. We will later discuss what happens when there is no appropriate *catch* block.

Next, we summarize what happens when no exception is thrown in a *try* block. If no exception (no value) is thrown in the *try* block, then after the *try* block is completed, program execution continues with the code after the *catch* block. In other words, if no exception is thrown, then the *catch* block is ignored. Most of the time when the program is executed, the *throw* statement will not be executed, and so in most cases, the code in the *try* block will run to completion and the code in the *catch* block will be ignored completely.

### try-throw-catch

This is the basic mechanism for throwing and catching exceptions. The **throw statement** throws the exception (a value). The **catch block** catches the exception (the value). When an exception is thrown, the *try* block ends and then the code in the *catch* block is executed. After the *catch* block is completed, the code after the *catch* block(s) is executed (provided the *catch* block has not ended the program or performed some other special action).

If no exception is thrown in the *try* block, then after the *try* block is completed, program execution continues with the code after the *catch* block(s). (In other words, if no exception is thrown, then the *catch* block(s) are ignored.)

**SYNTAX**

```
try
{
    Some_Statements
        < Either some code with a throw statement or a
            function invocation that might throw an
            exception>
    Some_More_Statements
}
catch(Type_Name e)
{
    < Code to be performed if a value of the
        catch-block parameter type is thrown in the
        try block>
}
```

**EXAMPLE**

See Display 16.2.

**SELF-TEST EXERCISES**

1. What output is produced by the following code?

```
int waitTime = 46;
try
{
    cout << "Try block entered.\n";
    if (waitTime > 30)
        throw waitTime;
    cout << "Leaving try block.\n";
}
catch(int thrownValue)
{
    cout << "Exception thrown with\n"
        << "waitTime equal to " << thrownValue << endl;
}
cout << "After catch block." << endl;
```

2. What would be the output produced by the code in Self-Test Exercise 1 if we make the following change? Change the line

```
int waitTime = 46;
```

to

```
int waitTime = 12;
```

3. In the code given in Self-Test Exercise 1, what is the *throw* statement?
4. What happens when a *throw* statement is executed? This is a general question. Tell what happens in general, not simply what happens in the code in Self-Test Question 1 or some other sample code.
5. In the code given in Self-Test Exercise 1, what is the *try* block?
6. In the code given in Self-Test Exercise 1, what is the *catch* block?
7. In the code given in Self-Test Exercise 1, what is the *catch-block parameter*?

## Defining Your Own Exception Classes

A *throw* statement can throw a value of any type. A common thing to do is to define a class whose objects can carry the precise kind of information you want thrown to the *catch* block. An even more important reason for defining a specialized exception class is so that you can have a different type to identify each possible kind of exceptional situation.

An exception class is just a class. What makes it an exception class is how it's used. Still, it pays to take some care in choosing an exception class's name and other details. Display 16.3 contains an example of a program with a programmer-defined exception class. This is just a toy program to illustrate some C++ details about exception handling. It uses much too much machinery for such a simple task, but it is an otherwise uncluttered example of some C++ details.

Notice the *throw* statement, reproduced in what follows:

```
throw NoMilk(donuts);
```

The part `NoMilk(donuts)` is an invocation of a constructor for the class `NoMilk`. The constructor takes one *int* argument (in this case `donuts`) and creates an object of the class `NoMilk`. That object is then "thrown."

## Multiple Throws and Catches

A *try* block can potentially throw any number of exception values, and they can be of differing types. In any one execution of the *try* block, only one exception will be thrown (since a thrown exception ends the execution of the *try* block), but different types of exception values can be thrown on different occasions when the *try* block is executed. Each *catch* block can only catch values of one type, but you can catch exception values of differing types by placing more than one *catch* block after a *try* block. For example, the program in Display 16.4 has two *catch* blocks after its *try* block.

Note that there is no parameter in the *catch* block for `DivideByZero`. If you do not need a parameter, you can simply list the type with no parameter.

**DISPLAY 16.3 Defining Your Own Exception Class**

```
1 #include <iostream>
2 using namespace std;
3
4 class NoMilk
5 {
6 public:
7     NoMilk();
8     NoMilk(int howMany);
9     int getDonuts();
10 private:
11     int count;
12 };
13
14 int main()
15 {
16     int donuts, milk;
17     double dpg;
18     try
19     {
20         cout << "Enter number of donuts:\n";
21         cin >> donuts;
22         cout << "Enter number of glasses of milk:\n";
23         cin >> milk;
24         if (milk <= 0)
25             throw NoMilk(donuts);
26         dpg = donuts/static_cast<double>(milk);
27         cout << donuts << " donuts.\n"
28             << milk << " glasses of milk.\n"
29             << "You have " << dpg
30             << " donuts for each glass of milk.\n";
31     }
32     catch(NoMilk e)
33     {
34         cout << e.getDonuts() << " donuts, and No Milk!\n"
35             << "Go buy some milk.\n";
36     }
37     cout << "End of program.";
38     return 0;
39 }
40
41 NoMilk::NoMilk()
42 {}
43 NoMilk::NoMilk(int howMany) : count(howMany)
44 {}
45
46 int NoMilk::getDonuts()
47 {
48     return count;
49 }
```

*This is just a toy example to learn C++ syntax. Do not take it as an example of good typical use of exception handling.*

*The sample dialogues are the same as in Display 16.2.*

**DISPLAY 16.4 Catching Multiple Exceptions (part 1 of 2)**

```

1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  class NegativeNumber
6  {
7  public:
8      NegativeNumber();
9      NegativeNumber(string takeMeToYourCatchBlock);
10     string getMessage();
11 private:
12     string message;
13 };
14
15 class DivideByZero
16 {};
17
18 int main()
19 {
20     int jemHadar, klingons;
21     double portion;
22
23 try
24 {
25     cout << "Enter number of JemHadar warriors:\n";
26     cin >> jemHadar;
27     if (jemHadar< 0)
28         throw NegativeNumber("JemHadar");
29
30     cout << "How many Klingon warriors do you have?\n";
31     cin >> klingons;
32     if (klingons< 0)
33         throw NegativeNumber("Klingons");
34     if (klingons != 0)
35         portion = jemHadar/static_cast<double>(klingons);
36     else
37         throw DivideByZero();
38     cout << "Each Klingon must fight "
39         << portion << " JemHadar.\n";
40 }
41 catch(NegativeNumber e)
42 {
43     cout << "Cannot have a negative number of "
44         << e.getMessage() << endl;
45 }
```

*Although not done here, exception classes can have their own interface and implementation files and can be put in a namespace.*  
*This is another toy example.*

(continued)

**DISPLAY 16.4 Catching Multiple Exceptions (part 2 of 2)**

```
46     catch ( DivideByZero )
47     {
48         cout << "Send for help.\n";
49     }
50
51     cout << "End of program.\n";
52     return 0;
53 }
54
55
56 NegativeNumber::NegativeNumber()
57 {}
58
59 NegativeNumber::NegativeNumber(string takeMeToYourCatchBlock)
60     : message(takeMeToYourCatchBlock)
61 {}
62
63 string NegativeNumber::getMessage()
64 {
65     return message;
66 }
```

**Sample Dialogue 1**

```
Enter number of JemHadar warriors:  
1000  
How many Klingon warriors do you have?  
500  
Each Klingon must fight 2.0 JemHadar.  
End of program
```

**Sample Dialogue 2**

```
Enter number of JemHadar warriors:  
-10  
Cannot have a negative number of JemHadar  
End of program.
```

**Sample Dialogue 3**

```
Enter number of JemHadar warriors:  
1000  
How many Klingon warriors do you have?  
0  
Send for help.  
End of program.
```

This case is discussed a bit more in the Programming Tip section entitled "Exception Classes Can Be Trivial."

### PITFALL **Catch the More Specific Exception First**

---

When catching multiple exceptions, the order of the *catch* blocks can be important. When an exception value is thrown in a *try* block, the following *catch* blocks are tried in order, and the first one that matches the type of the exception thrown is the one that is executed.

For example, the following is a special kind of *catch* block that will catch a thrown value of any type:

```
catch(...)  
{  
    <Place whatever you want in here>  
}
```

The three dots do not stand for something omitted. You actually type in those three dots in your program. This makes a good default *catch* block to place after all other *catch* blocks. For example, we could add it to the *catch* blocks in Display 16.4 as follows:

```
catch(NegativeNumber e)  
{  
    cout << "Cannot have a negative number of "  
        << e.getMessage() << endl;  
}  
catch(DivideByZero)  
{  
    cout << "Send for help.\n";  
}  
catch(...)  
{  
    cout << "Unexplained exception.\n";  
}
```

However, it only makes sense to place this default *catch* block at the end of a list of *catch* blocks. For example, suppose we instead used:

```
catch(NegativeNumber e)  
{  
    cout << "Cannot have a negative number of "  
        << e.getMessage() << endl;  
}  
catch(...)  
{  
    cout << "Unexplained exception.\n";  
}  
catch(DivideByZero)
```

```
{  
    cout << "Send for help.\n";  
}
```

With this second ordering, an exception (a thrown value) of type `NegativeNumber` will be caught by the `NegativeNumber` *catch* block, as it should be. However, if a value of type `DivideByZero` were thrown, it would be caught by the block that starts `catch(...)`. So, the `DivideByZero` *catch* block could never be reached. Fortunately, most compilers tell you if you make this sort of mistake. ■

### ■ PROGRAMMING TIP Exception Classes Can Be Trivial

Here we reproduce the definition of the exception class `DivideByZero` from Display 16.4:

```
class DivideByZero  
{};
```

This exception class has no member variables and no member functions (other than the default constructor). It has nothing but its name, but that is useful enough. Throwing an object of the class `DivideByZero` can activate the appropriate *catch* block, as it does in Display 16.4.

When using a trivial exception class, you normally do not have anything you can do with the exception (the thrown value) once it gets to the *catch* block. The exception is just being used to get you to the *catch* block. Thus, you can omit the *catch*-block parameter. (You can omit the *catch*-block parameter anytime you do not need it, whether the exception type is trivial or not.) ■

### Throwing an Exception in a Function

Sometimes it makes sense to delay handling an exception. For example, you might have a function with code that throws an exception if there is an attempt to divide by zero, but you may not want to catch the exception in that function. Perhaps some programs that use that function should simply end if the exception is thrown, and other programs that use the function should do something else. So you would not know what to do with the exception if you caught it inside the function. In these cases, it makes sense to not catch the exception in the function definition, but instead to have any program (or other code) that uses the function place the function invocation in a *try* block and catch the exception in a *catch* block that follows that *try* block.

Look at the program in Display 16.5. It has a *try* block, but there is no *throw* statement visible in the *try* block. The statement that does the throwing in that program is

```
if (bottom == 0)  
    throw DivideByZero();
```

**DISPLAY 16.5 Throwing an Exception Inside a Function (part 1 of 2)**

```
1 #include <iostream>
2 #include <cstdlib>
3 using namespace std;
4
5 class DivideByZero
6 {};
7
8 double safeDivide(int top, int bottom) throw (DivideByZero);
9
10 int main()
11 {
12     int numerator;
13     int denominator;
14     double quotient;
15     cout << "Enter numerator:\n";
16     cin >> numerator;
17     cout << "Enter denominator:\n";
18     cin >> denominator;
19
20     try
21     {
22         quotient = safeDivide(numerator, denominator);
23     }
24     catch(DivideByZero)
25     {
26         cout << "Error: Division by zero!\n"
27             << "Program aborting.\n";
28         exit(0);
29     }
30
31     cout << numerator << "/" << denominator
32         << " = " << quotient << endl;
33
34     cout << "End of program.\n";
35     return 0;
36 }
37
38
39 double safeDivide(int top, int bottom) throw (DivideByZero)
40 {
41     if (bottom == 0)
42         throw DivideByZero();
43
44     return top/static_cast<double>(bottom);
45 }
```

(continued)

**DISPLAY 16.5 Throwing an Exception Inside a Function (part 2 of 2)****Sample Dialogue 1**

```
Enter numerator:
```

```
5
```

```
Enter denominator:
```

```
10
```

```
5/10 = 0.5
```

```
End of Program.
```

**Sample Dialogue 2**

```
Enter numerator:
```

```
5
```

```
Enter denominator:
```

```
0
```

```
Error: Division by zero!
```

```
Program aborting.
```

This statement is not visible in the *try* block. However, it is in the *try* block in terms of program execution, because it is in the definition of the function `safeDivide` and there is an invocation of `safeDivide` in the *try* block.

## Exception Specification

If a function does not catch an exception, it should at least warn programmers that any invocation of the function might possibly throw an exception. If there are exceptions that might be thrown, but not caught, in the function definition, then those exception types should be listed in an **exception specification**, which is illustrated by the following function declaration from Display 16.5:

```
double safeDivide(int top, int bottom) throw (DivideByZero);
```

As illustrated in Display 16.5, the exception specification should appear in both the function declaration and the function definition. If a function has more than one function declaration, then all the function declarations must have identical exception specifications. The exception specification for a function is also sometimes called the **throw list**.

If there is more than one possible exception that can be thrown in the function definition, then the exception types are separated by commas, as illustrated here:

```
void someFunction( ) throw (DivideByZero, OtherException);
```

All exception types listed in the exception specification are treated normally. When we say the exception is treated normally, we mean it is treated as we have described before this subsection. In particular, you can place the function invocation in a *try* block followed by a *catch* block to catch that type of exception, and if the function throws the exception (and does not catch it inside the function), then the *catch* block following the *try* block will catch the exception. If there is no exception specification (no throw list) at all (not even an empty one), then it is the same as if all possible exception types were listed in the exception specification; that is, any exception that is thrown is treated normally.

What happens when an exception is thrown in a function but is not listed in the exception specification (and not caught inside the function)? In that case, the program ends. In particular, notice that if an exception is thrown in a function but is not listed in the exception specification (and not caught inside the function), then it will not be caught by any *catch* block, but instead your program will end. Remember, if there is no specification list at all, not even an empty one, then it is the same as if all exceptions were listed in the specification list, and so throwing an exception will not end the program in the way described in this paragraph.

Keep in mind that the exception specification is for exceptions that “get outside” the function. If they do not get outside the function, they do not belong in the exception specification. If they get outside the function, they belong in the exception specification no matter where they originate. If an exception is thrown in a *try* block that is inside a function definition and is caught in a *catch* block inside the function definition, then its type need not be listed in the exception specification. If a function definition includes an invocation of another function and that other function can throw an exception that is not caught, then the type of the exception should be placed in the exception specification.

To say that a function should not throw any exceptions that are not caught inside the function, you use an empty exception specification like so:

```
void someFunction( ) throw ( );
```

By way of summary:

```
void someFunction( ) throw (DivideByZero, OtherException);  
//Exceptions of type DivideByZero or OtherException are  
//treated normally. All other exceptions end the program  
//if not caught in the function body.
```

```
void someFunction( ) throw ( );  
//Empty exception list; all exceptions end the  
//program if thrown but not caught in the function body.
```

```
void someFunction( );  
//All exceptions of all types treated normally.
```

Keep in mind that an object of a derived class<sup>1</sup> is also an object of its base class. So, if D is a derived class of class B and B is in the exception specification, then a thrown object of class D will be treated normally, since it is an object of class B and B is in the exception specification. However, no automatic type conversions are done. If *double* is in the exception specification, that does not account for throwing an *int* value. You would need to include both *int* and *double* in the exception specification.

One final warning: Not all compilers treat the exception specification as they are supposed to. Some compilers essentially treat the exception specification as a comment, and so with those compilers, the exception specification has no effect on your code. This is another reason to place all exceptions that might be thrown by your functions in the exception specification. This way all compilers will treat your exceptions the same way. Of course, you could get the same compiler consistency by not having any exception specification at all, but then your program would not be as well documented and you would not get the extra error checking provided by compilers that do use the exception specification. With a compiler that does process the exception specification, your program will terminate as soon as it throws an exception that you did not anticipate. (Note that this is a run-time behavior, but which run-time behavior you get depends on your compiler.)

Warning!

### PITFALL Exception Specification in Derived Classes

When you redefine or override a function definition in a derived class, it should have the same exception specification as it had in the base class, or it should have an exception specification whose exceptions are a subset of those in the base class exception specification. Put another way, when you redefine or override a function definition, you cannot add any exceptions to the exception specification (but you can delete some exceptions if you want). This makes sense, since an object of the derived class can be used anywhere an object of the base class can be used, and so a redefined or overwritten function must fit any code written for an object of the base class. ■

### SELF-TEST EXERCISES

- What is the output produced by the following program?

```
#include <iostream>
using namespace std;
void sampleFunction(double test) throw (int);
```

---

<sup>1</sup> If you have not yet learned about derived classes, you can safely ignore the remarks about them.

```

int main()
{
    try
    {
        cout << "Trying.\n";
        sampleFunction(98.6);
        cout << "Trying after call.\n";
    }
    catch(int)
    {
        cout << "Catching.\n";
    }
    cout << "End of program.\n";
    return 0;
}
void sampleFunction(double test) throw (int)
{
    cout << "Starting sampleFunction.\n";
    if (test < 100)
        throw 42;
}

```

9. What is the output produced by the program in Self-Test Exercise 8 if the following change were made to the program? Change

`sampleFunction(98.6);`

in the *try* block to

`sampleFunction(212);`

## 16.2 PROGRAMMING TECHNIQUES FOR EXCEPTION HANDLING

*Only use this in exceptional circumstances.*

WARREN PEACE, *The Lieutenant's Tools*

So far, we have shown you lots of code that explains how exception handling works in C++, but we have not yet shown even one example of a program that makes good and realistic use of exception handling. However, now that you know the mechanics of exception handling, this section can go on to explain exception-handling techniques.

### When to Throw an Exception

We have given some very simple code in order to illustrate the basic concepts of exception handling. However, our examples were unrealistically simple. A more complicated but better guideline is to separate throwing an exception

and catching the exception into separate functions. In most cases, you should include any *throw* statement within a function definition, list the exception in the exception specification for that function, and place the *catch* clause in a *different function*. Thus, the preferred use of the *try-throw-catch* triad is as illustrated here:

```
void functionA() throw (MyException)
{
    .
    .
    .
    throw MyException(<Maybe an argument>);
    .
    .
    .
}
```

Then, in *some other function* (perhaps even some other function in some other file), you have

```
void functionB()
{
    .
    .
    .

    try
    {
        .
        .
        .

        functionA();

        .
        .
        .

    }
    catch(MyException e)
    {
        <Handle exception>
    }
    .
    .
    .

}

}
```

Moreover, even this kind of use of a *throw* statement should be reserved for cases in which it is unavoidable. If you can easily handle a problem in some other way, do not throw an exception. Reserve *throw* statements for situations in which the way the exceptional condition is handled depends on how and where the function is used. If the way that the exceptional condition is handled depends on how and where the function is invoked, then the best

### When to Throw an Exception

For the most part, *throw* statements should be used within functions and listed in an exception specification for the function. Moreover, they should be reserved for situations in which the way the exceptional condition is handled depends on how and where the function is used. If the way that the exceptional condition is handled depends on how and where the function is invoked, then the best thing to do is to let the programmer who invokes the function handle the exception. In all other situations, it is almost always preferable to avoid throwing an exception.

thing to do is to let the programmer who invokes the function handle the exception. In all other situations, it is almost always preferable to avoid throwing exceptions.

### PITFALL **Uncaught Exceptions**

Every exception that is thrown by your code should be caught someplace in your code. If an exception is thrown but not caught anywhere, your program will end. ■

### PITFALL **Nested try-catch Blocks**

You can place a *try* block and following *catch* blocks inside a larger *try* block or inside a larger *catch* block. In rare cases, this may be useful, but if you are tempted to do this, you should suspect that there is a nicer way to organize your program. It is almost always better to place the inner *try-catch* blocks inside a function definition and place an invocation of the function in the outer *try* or *catch* block (or maybe just eliminate one or more *try* blocks completely).

If you place a *try* block and following *catch* blocks inside a larger *try* block, and an exception is thrown in the inner *try* block but not caught in the inner *try-catch* blocks, then the exception is thrown to the outer *try* block for processing and might be caught there. ■

### PITFALL **Overuse of Exceptions**

Exceptions allow you to write programs whose flow of control is so involved that it is almost impossible to understand the program. Moreover, this is not hard to do. Throwing an exception allows you to transfer flow of control from

anyplace in your program to almost anywhere else in your program. In the early days of programming, this sort of unrestricted flow of control was allowed via a construct known as a *goto*. Programming experts now agree that such unrestricted flow of control is very poor programming style. Exceptions allow you to revert to these bad old days of unrestricted flow of control. Exceptions should be used sparingly and only in certain ways. A good rule is the following: If you are tempted to include a *throw* statement, then think about how you might write your program or class definition without this *throw* statement. If you think of an alternative that produces reasonable code, then you probably do not want to include the *throw* statement. ■

## Exception Class Hierarchies

It can be very useful to define a hierarchy of exception classes. For example, you might have an `ArithmaticError` exception class and then define an exception class `DivideByZeroError` that is a derived class of `ArithmaticError`. Since a `DivideByZeroError` is an `ArithmaticError`, every *catch* block for an `ArithmaticError` will catch a `DivideByZeroError`. If you list `ArithmaticError` in an exception specification, then you have, in effect, also added `DivideByZeroError` to the exception specification, whether or not you list `DivideByZeroError` by name in the exception specification.



## Testing for Available Memory

In Chapter 13, we created new dynamic variables with code such as the following:

```
struct Node
{
    int data;
    Node *link;
};

typedef Node* NodePtr;

...
NodePtr pointer = new Node;
```

This works fine as long as there is sufficient memory available to create the new node. But, what happens if there is not sufficient memory? If there is not sufficient memory to create the node, then a `bad_alloc` exception is thrown. The type `bad_alloc` is part of the C++ language. You do not need to define it.

Since `new` will throw a `bad_alloc` exception when there is not enough memory to create the node, you can check for running out of memory as follows:

```
try
{
    NodePtr pointer = new Node;
}
```

```
catch (badAlloc)
{
    cout << "Ran out of memory!";
}
```

Of course, you can do other things besides simply giving a warning message, but the details of what you do will depend on your particular programming task.

### Rethrowing an Exception

It is legal to throw an exception within a *catch* block. In rare cases, you may want to catch an exception and then, depending on the details, decide to throw the same or a different exception for handling farther up the chain of exception-handling blocks.

### SELF-TEST EXERCISES

10. What happens when an exception is never caught?
11. Can you nest a *try* block inside another *try* block?

### CHAPTER SUMMARY

- Exception handling allows you to design and code the normal case for your program separately from the code that handles exceptional situations.
- An exception can be thrown in a *try* block. Alternatively, an exception can be thrown in a function definition that does not include a *try* block (or does not include a *catch* block to catch that type of exception). In this case, an invocation of the function can be placed in a *try* block.
- An exception is caught in a *catch* block.
- A *try* block may be followed by more than one *catch* block. In this case, always list the *catch* block for a more specific exception class before the *catch* block for a more general exception class.
- Do not overuse exceptions.

### Answers to Self-Test Exercises

1. Try block entered.  
Exception thrown with  
waitTime equal to 46  
After catch block.

2. Try block entered.  
Leaving try block.  
After catch block.

3. `throw waitTime;`

Note that the following is an *if* statement, not a *throw* statement, even though it contains a *throw* statement:

```
if (waitTime > 30)
    throw waitTime;
```

4. When a *throw* statement is executed, that is the end of the enclosing *try* block. No other statements in the *try* block are executed, and control passes to the following *catch* block(s). When we say control passes to the following *catch* block, we mean that the value thrown is plugged in for the *catch*-block parameter (if any), and the code in the *catch* block is executed.

5. `try`

```
{  
    cout << "Try block entered.";  
    if (waitTime > 30)  
        throw (waitTime);  
    cout << "Leaving try block.";  
}
```

6. `catch(int thrownValue)`

```
{  
    cout << "Exception thrown with\n"  
        << "waitTime equal to" << thrownValue << endl;  
}
```

7. `thrownValue` is the *catch*-block parameter.

8. Trying.

```
Starting sampleFunction.  
Catching.  
End of program.
```

9. Trying.

```
Starting sampleFunction.  
Trying after call.  
End of program.
```

10. If an exception is not caught anywhere, then your program ends.

11. Yes, you can have a *try* block and corresponding *catch* blocks inside another larger *try* block. However, it would probably be better to place the inner *try* and *catch* blocks in a function definition and place an invocation of the function in the larger *try* block.

## PRACTICE PROGRAMS

*Practice Programs can generally be solved with a short program that directly applies the programming principles presented in this chapter.*



1. A function that returns a special error code is often better implemented by throwing an exception instead. This way, the error code cannot be ignored or mistaken for valid data. The following class maintains an account balance.

```
class Account
{
    private:
        double balance;
    public:
        Account()
        {
            balance = 0;
        }
        Account(double initialDeposit)
        {
            balance = initialDeposit;
        }
        double getBalance()
        {
            return balance;
        }
        // returns new balance or -1 if error
        double deposit(double amount)
        {
            if (amount > 0)
                balance += amount;
            else
                return -1; // Code indicating error
            return balance;
        }
        // returns new balance or -1 if invalid amount
        double withdraw(double amount)
        {
            if ((amount > balance) || (amount < 0))
                return -1;
            else
                balance -= amount;
            return balance;
        }
};
```

Rewrite the class so that it throws appropriate exceptions instead of returning -1 as an error code. Write test code that attempts to withdraw and deposit invalid amounts and catches the exceptions that are thrown.

2. The Standard Template Library includes a class named `exception` that is the parent class for any exception thrown by an STL function. Therefore, any exception can be caught by this class. The following code sets up a *try-catch* block for STL exceptions:

```
#include <iostream>
#include <string>
#include <exception>
using namespace std;

int main()
{
    string s = "hello";
    try
    {
        cout << "No exception thrown." << endl;
    }
    catch (exception& e)
    {
        cout << "Exception caught: " <<
            e.what() << endl;
    }
    return 0;
}
```

Modify the code so that an exception is thrown in the `try` block. You could try accessing an invalid index in a string using the `at` member function.

## PROGRAMMING PROJECTS

*Programming Projects require more problem-solving than Practice Programs and can usually be solved many different ways. Visit [www.myprogramminglab.com](http://www.myprogramminglab.com) to complete many of these Programming Projects online and get instant feedback.*

1. Write a function to convert a hexadecimal number given as a `String` to an integer value. If the value to be converted is not a valid hexadecimal number, throw an `InvalidNumberException` before attempting the conversion. You will need to develop both the function and the exception class. Write a driver program to test valid and invalid input to your function.

*All men are mortal.*  
*Aristotle is a man.*  
*Therefore, Aristotle is mortal.*  
*All X's are Y.*  
*Z is an X.*  
*Therefore, Z is Y.*  
*All cats are mischievous.*  
*Garfield is a cat.*  
*Therefore, Garfield is mischievous.*

#### A SHORT LESSON ON SYLLOGISMS

---

## INTRODUCTION

This chapter discusses C++ templates. Templates allow you to define functions and classes that have parameters for type names. This will allow you to design functions that can be used with arguments of different types and to define classes that are much more general than those you have seen before this chapter.

## PREREQUISITES

Section 17.1 uses material from Chapters 2 through 5 and Sections 7.1, 7.2, and 7.3 of Chapter 7. It does not use any material on classes. Section 17.2 uses material from Chapters 2 through 7 and 10 through 12.

## 17.1 TEMPLATES FOR ALGORITHM ABSTRACTION

Many of our previous C++ function definitions have an underlying algorithm that is much more general than the algorithm we gave in the function definition. For example, consider the function `swapValues`, which we first discussed in Chapter 5. For reference, we now repeat the function definition:

```
void swapValues(int& variable1, int& variable2)
{
    int temp;

    temp = variable1;
    variable1 = variable2;
    variable2 = temp;
}
```

Notice that the function `swapValues` applies only to variables of type `int`. Yet the algorithm given in the function body could just as well be used to swap the values in two variables of type `char`. If we want to also use the function

`swapValues` with variables of type `char`, we can overload the function name by adding the following definition:

```
void swapValues(char& variable1, char& variable2)
{
    char temp;

    temp = variable1;
    variable1 = variable2;
    variable2 = temp;
}
```

But there is something inefficient and unsatisfying about these two definitions of the `swapValues` function: They are almost identical. The only difference is that one definition uses the type `int` in three places and the other uses the type `char` in the same three places. Proceeding in this way, if we wanted to have the function `swapValues` apply to pairs of variables of type `double`, we would have to write a third almost identical function definition. If we wanted to apply `swapValues` to still more types, the number of almost identical function definitions would be even larger. This would require a good deal of typing and would clutter up our code with lots of definitions that look identical. We should be able to say that the following function definition applies to variables of any type:

```
void swapValues(Type_Of_The_Variables& variable1,
                Type_Of_The_Variables& variable2)
{
    Type_Of_The_Variables temp;

    temp = variable1;
    variable1 = variable2;
    variable2 = temp;
}
```

As we will see, something like this is possible. We can define one function that applies to all types of variables, although the syntax is a bit different from what we have shown above. That syntax is described in the next subsection.

## Templates for Functions

Display 17.1 shows a C++ template for the function `swapValues`. This function template allows you to swap the values of any two variables, of any type, as long as the two variables have the same type. The definition and the function declaration begin with the line

```
template<class T>
```

This is often called the **template prefix**, and it tells the compiler that the definition or function declaration that follows is a **template** and that `T` is a

**type parameter.** In this context, the word *class* actually means *type*.<sup>1</sup> As we will see, the type parameter T can be replaced by any type, whether the type is a class or not. Within the body of the function definition, the type parameter T is used just like any other type.

The function template definition is, in effect, a large collection of function definitions. For the function template for swapValues shown in Display 17.1, there is, in effect, one function definition for each possible type name. Each of these definitions is obtained by replacing the type parameter T with a type name. For example, the function definition that follows is obtained by replacing T with the type name *double*:

```
void swapValues(double& variable1, double& variable2)
{
    double temp;
    temp = variable1;
    variable1 = variable2;
    variable2 = temp;
}
```

A template  
overloads the  
function name

Another definition for swapValues is obtained by replacing the type parameter T in the function template with the type name *int*. Yet another definition is obtained by replacing the type parameter T with *char*. The one function template shown in Display 17.1 overloads the function name swapValues so that there is a slightly different function definition for every possible type.

The compiler will not literally produce definitions for every possible type for the function name swapValues, but it will behave exactly as if it had produced all those function definitions. A separate definition will be produced for each different type for which you use the template, but not for any types you do not use. Only one definition is generated for a single type regardless of the number of times you use the template for that type. Notice that the function swapValues is called twice in Display 17.1: One time the arguments are of type *int* and the other time the arguments are of type *char*.

Consider the following function call from Display 17.1:

```
swapValues(integer1, integer2);
```

When the C++ compiler gets to this function call, it notices the types of the arguments—in this case *int*—and then it uses the template to produce a function definition with the type parameter T replaced with the type name *int*. Similarly, when the compiler sees the function call

```
swapValues(symbol1, symbol2);
```

---

<sup>1</sup> In fact, the ANSI standard provides that you may use the keyword *typename* instead of *class* in the template prefix. Although we agree that using *typename* makes more sense than using *class*, the use of *class* is a firmly established tradition, and so we use *class* for the sake of consistency with most other programmers and authors.

**DISPLAY17.1 A Function Template**

```
1 //Program to demonstrate a function template.
2 #include <iostream>
3 using namespace std;
4 //Interchanges the values of variable1 and variable2.
5 template<class T>
6 void swapValues(T& variable1, T& variable2)
7 {
8     T temp;
9
10    temp = variable1;
11    variable1 = variable2;
12    variable2 = temp;
13 }
14 int main( )
15 {
16     int integer1 = 1, integer2 = 2;
17     cout << "Original integer values are "
18         << integer1 << " " << integer2 << endl;
19     swapValues(integer1, integer2);
20     cout << "Swapped integer values are "
21         << integer1 << " " << integer2 << endl;
22
23     char symbol1 = 'A', symbol2 = 'B';
24     cout << "Original character values are "
25         << symbol1 << " " << symbol2 << endl;
26     swapValues(symbol1, symbol2);
27     cout << "Swapped character values are "
28         << symbol1 << " " << symbol2 << endl;
29 }
```

**Output**

```
Original integer values are 1 2
Swapped integer values are 2 1
Original character values are A B
Swapped character values are B A
```

it notices the types of the arguments—in this case *char*—and then it uses the template to produce a function definition with the type parameter *T* replaced with the type name *char*.

Notice that you need not do anything special when you call a function that is defined with a function template; you call it just as you would any

Calling a  
function  
template

other function. The compiler does all the work of producing the function definition from the function template.

Notice that in Display 17.1 we placed the function template definition before the `main` part of the program, and we used no template function declaration. A function template may have a function declaration, just like an ordinary function. You may (or may not) be able to place the function declaration and definition for a function template in the same locations that you place function declarations and definitions for ordinary functions. However, some compilers do not support template function declarations and do not support separate compilation of template functions. When these are supported, the details can be messy and can vary from one compiler to another. Your safest strategy is to not use template function declarations and to be sure the function template definition appears in the same file in which it is used and appears before the function template is used.

We said that a function template definition should appear in the same file as the file that uses the template function (that is, the same file as the file that has an invocation of the template function). However, the function template definition can appear via a `#include` directive. You can give the function template definition in one file and then `#include` that file in a file that uses the template function. That is the cleanest and safest general strategy. However, even that may not work on some compilers. If it does not work, consult a local expert.

Although we will not be using template function declarations in our code, we will describe them and give examples of them for the benefit of readers whose compilers support the use of these function declarations.

In the function template in Display 17.1, we used the letter `T` as the parameter for the type. This is traditional but is not required by the C++ language. The type parameter can be any identifier (other than a keyword). `T` is a good name for the type parameter, but sometimes other names may work better. For example, the function template for `swapValues` given in Display 17.1 is equivalent to the following:

```
template<class VariableType>
void swapValues(VariableType& variable1,
                VariableType& variable2)
{
    VariableType temp;
    temp = variable1;
    variable1 = variable2;
    variable2 = temp;
}
```

More than one  
type parameter

It is possible to have function templates that have more than one type parameter. For example, a function template with two type parameters named `T1` and `T2` would begin as follows:

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

However, most function templates require only one type parameter. You cannot have unused template parameters; that is, each template parameter must be used in your template function.

## PITFALL Compiler Complications



VideoNote  
Issues Compiling Programs  
with Templates

C++ does not allow you to separate interface (header) and implementation files for template definitions in the usual way, so you need to include your template definition with your code that uses it. As usual, at least the function declaration must precede any use of the template function. This is because the header can't correctly match the implementation when the type is unknown.

Your safest strategy is not to use template function declarations and to be sure the function template definition appears in the same file in which it is used and appears before the function template is called. However, the function template definition can appear via a `#include` directive. You can give the function template definition in one file and then `#include` that file in a file that uses the template function.

Another common technique is to put your definition and implementation, all in the header file. If you use this technique, then you would only have a header (.h) file and no implementation (.cpp) file. Sometimes the .hpp file extension is used when all of the code is in the header file. Finally, an alternate approach is to include the implementation (.cpp) file for your template class instead of the header file (.h).

Some C++ compilers have additional special requirements for using templates. If you have trouble compiling your templates, check your manuals or check with a local expert. You may need to set special options or rearrange the way you order the template definitions and the other items in your files. ■

### Function Template

The function definition and the function declaration for a function template are each prefaced with the following:

`template<class Type_Parameter>`

The function declaration (if used) and definition are the same as any ordinary function declaration and definition, except that the `Type_Parameter` can be used in place of a type.

For example, the following is a function declaration for a function template:

```
template<class T>
void showStuff(int stuff1, T stuff2, T stuff3);
```

*(continued)*

The definition for this function template might be as follows:

```
template<class T>
void showStuff(int stuff1, T stuff2, T stuff3)
{
    cout << stuff1 << endl
        << stuff2 << endl
        << stuff3 << endl;
}
```

The function template given in this example is equivalent to having one function declaration and one function definition for each possible type name. The type name is substituted for the type parameter (which is *T* in the example above). For instance, consider the following function call:

```
showStuff(2, 3.3, 4.4);
```

When this function call is executed, the compiler uses the function definition obtained by replacing *T* with the type name *double*. A separate definition will be produced for each different type for which you use the template but not for any types you do not use. Only one definition is generated for a specific type regardless of the number of times you use the template.

## SELF-TEST EXERCISES

1. Write a function template named `maximum`. The function takes two values of the same type as its arguments and returns the larger of the two arguments (or either value if they are equal). Give both the function declaration and the function definition for the template. You will use the operator `<` in your definition. Therefore, this function template will apply only to types for which `<` is defined. Write a comment for the function declaration that explains this restriction.
2. We have used three kinds of absolute value function: `abs`, `labs`, and `fabs`. These functions differ only in the type of their argument. It might be better to have a function template for the absolute value function. Give a function template for an absolute value function called `absolute`. The template will apply only to types for which `<` is defined, for which the unary negation operator is defined, and for which the constant `0` can be used in a comparison with a value of that type. Thus, the function `absolute` can be called with any of the number types, such as `int`, `long`, and `double`. Give both the function declaration and the function definition for the template.
3. Define or characterize the template facility for C++.

## 17.2 TEMPLATES FOR DATA ABSTRACTION

*Equal wealth and equal opportunities of culture . . . have simply made us all members of one class.*

EDWARD BELLAMY, *Looking Backward: 2000–1887*

As you saw in the previous section, function definitions can be made more general by using templates. In this section, you will see that templates can also make class definitions more general.

### Syntax for Class Templates

The syntax for class templates is basically the same as that for function templates. The following is placed before the template definition:

```
template<class T>
```

The type parameter *T* is used in the class definition just like any other type. As with function templates, the type parameter *T* represents a type that can be any type at all; the type parameter does not have to be replaced with a class type. As with function templates, you may use any (nonkeyword) identifier instead of *T*.

Type parameter

For example, the following is a class template. An object of this class contains a pair of values of type *T*; if *T* is *int*, the object values are pairs of integers, if *T* is *char*, the object values are pairs of characters, and so on.

```
//Class for a pair of values of type T:
template<class T>
class Pair
{
public:
    Pair();

    Pair(T firstValue, T secondValue);

    void setElement(int position, T value);
    //Precondition: position is 1 or 2.
    //Postcondition:
    //The position indicated has been set to value.

    T getElement(int position) const;
    //Precondition: position is 1 or 2.
    //Returns the value in the position indicated.

private:
    T first;
    T second;
};
```

Once the class template is defined, you can declare objects of this class. The declaration must specify what type is to be filled in for *T*. For example, the

Declaring objects

following code declares the object `score` so it can record a pair of integers and declares the object `seats` so it can record a pair of characters:

```
Pair<int> score;
Pair<char> seats;
```

The objects are then used just like any other objects. For example, the following sets the score to be 3 for the first team and 0 for the second team:

```
score.setElement(1, 3);
score.setElement(2, 0);
```

#### Defining member functions

The member functions for a class template are defined the same way as member functions for ordinary classes. The only difference is that the member function definitions are themselves templates. For example, the following are appropriate definitions for the member function `setElement` and for the constructor with two arguments:

```
//Uses iostream and cstdlib:
template<class T>
void Pair<T>::setElement(int position, T value)
{
    if (position == 1)
        first = value;
    else if (position == 2)
        second = value;
    else
    {
        cout << "Error: Illegal pair position.\n";
        exit(1);
    }
}

template<class T>
Pair<T>::Pair(T firstValue, T secondValue)
    : first(firstValue), second(secondValue)
{
    //Body intentionally empty.
}
```

Notice that the class name before the scope resolution operator is `Pair<T>`, not simply `Pair`.

The name of a class template may be used as the type for a function parameter. For example, the following is a possible declaration for a function with a parameter for a pair of integers:

```
int addUp(const Pair<int>& thePair);
//Returns the sum of the two integers in thePair.
```

### Class Template Syntax

The class definition and the definitions of the member functions are prefaced with the following:

```
template<class Type_Parameter>
```

The class and member function definitions are then the same as for any ordinary class, except that the *Type\_Parameter* can be used in place of a type.

For example, the following is the beginning of a class template definition:

```
template<class T>
class Pair
{
public:
    Pair();
    Pair(T firstValue, T secondValue);
    void setElement(int position, T value);
    . . .
```

Member functions and overloaded operators are then defined as function templates. For example, the definition of a function definition for the sample class template above could begin as follows:

```
template<class T>
void Pair<T>::setElement(int position, T value)
{
    . . .
```

Note that we specified the type, in this case *int*, that is to be filled in for the type parameter *T*.

You can even use a class template within a function template. For example, rather than defining the specialized function *addUp* given above, you could instead define a function template as follows so that the function applies to all kinds of numbers:

```
template<class T>
T addUp(const Pair<T>& thePair);
//Precondition: The operator + is defined for values of type T.
//Returns the sum of the two values in thePair.
```

# The ASCII Character Set

Only the printable characters are shown. Character number 32 is the blank.

|    |    |    |   |     |   |     |   |
|----|----|----|---|-----|---|-----|---|
| 32 |    | 56 | 8 | 80  | P | 104 | h |
| 33 | !  | 57 | 9 | 81  | Q | 105 | i |
| 34 | "  | 58 | : | 82  | R | 106 | j |
| 35 | #  | 59 | ; | 83  | S | 107 | k |
| 36 | \$ | 60 | < | 84  | T | 108 | l |
| 37 | %  | 61 | = | 85  | U | 109 | m |
| 38 | &  | 62 | > | 86  | V | 110 | n |
| 39 | '  | 63 | ? | 87  | W | 111 | o |
| 40 | (  | 64 | @ | 88  | X | 112 | p |
| 41 | )  | 65 | A | 89  | Y | 113 | q |
| 42 | *  | 66 | B | 90  | Z | 114 | r |
| 43 | +  | 67 | C | 91  | [ | 115 | s |
| 44 | ,  | 68 | D | 92  | \ | 116 | t |
| 45 | -  | 69 | E | 93  | ] | 117 | u |
| 46 | .  | 70 | F | 94  | ^ | 118 | v |
| 47 | /  | 71 | G | 95  | _ | 119 | w |
| 48 | 0  | 72 | H | 96  | ' | 120 | x |
| 49 | 1  | 73 | I | 97  | a | 121 | y |
| 50 | 2  | 74 | J | 98  | b | 122 | z |
| 51 | 3  | 75 | K | 99  | c | 123 | { |
| 52 | 4  | 76 | L | 100 | d | 124 |   |
| 53 | 5  | 77 | M | 101 | e | 125 | } |
| 54 | 6  | 78 | N | 102 | f | 126 | ~ |
| 55 | 7  | 79 | O | 103 | g |     |   |

# Some Library Functions



The following lists are organized according to what the function is used for, rather than what library it is in. The function declaration gives the number and types of arguments as well as the type of the value returned. In most cases, the function declarations give only the type of the parameter and do not give a parameter name. (See the section "Alternate Form for Function Declarations" in Chapter 4 for an explanation of this kind of function declaration.)

## Arithmetic Functions

| Function Declaration                     | Description                                                                 | Header File |
|------------------------------------------|-----------------------------------------------------------------------------|-------------|
| <code>int abs(int);</code>               | Absolute value                                                              | cstdlib     |
| <code>long labs(long);</code>            | Absolute value                                                              | cstdlib     |
| <code>double fabs(double);</code>        | Absolute value                                                              | cmath       |
| <code>double sqrt(double);</code>        | Square root                                                                 | cmath       |
| <code>double pow(double, double);</code> | Returns the first argument raised to the power of the second argument.      | cmath       |
| <code>double exp(double);</code>         | Returns e (base of the natural logarithm) to the power of its argument.     | cmath       |
| <code>double log(double);</code>         | Natural logarithm ( $\ln$ )                                                 | cmath       |
| <code>double log10(double);</code>       | Base 10 logarithm                                                           | cmath       |
| <code>double ceil(double);</code>        | Returns the smallest integer that is greater than or equal to its argument. | cmath       |
| <code>double floor(double);</code>       | Returns the largest integer that is less than or equal to its argument.     | cmath       |

## Input and Output Member Functions

| Form of a Function Call                                                  | Description                                                                                                                                                                                                                                                                                                         | Header File                                   |
|--------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------|
| <code>Stream_Var.open<br/>(External_File_Name);</code>                   | Connects the file with the <i>External_File_Name</i> to the stream named by the <i>Stream_Var</i> . The <i>External_File_Name</i> is a string value.                                                                                                                                                                | <code>fstream</code>                          |
| <code>Stream_Var.fail( );</code>                                         | Returns <i>true</i> if the previous operation (such as <code>open</code> ) on the stream <i>Stream_Var</i> has failed.                                                                                                                                                                                              | <code>fstream</code> or <code>iostream</code> |
| <code>Stream_Var.close( );</code>                                        | Disconnects the stream <i>Stream_Var</i> from the file it is connected to.                                                                                                                                                                                                                                          | <code>fstream</code>                          |
| <code>Stream_Var.bad( );</code>                                          | Returns <i>true</i> if the stream <i>Stream_Var</i> is corrupted.                                                                                                                                                                                                                                                   | <code>fstream</code> or <code>iostream</code> |
| <code>Stream_Var.eof( );</code>                                          | Returns <i>true</i> if the program has attempted to read beyond the last character in the file connected to the input stream <i>Stream_Var</i> . Otherwise, it returns <i>false</i> .                                                                                                                               | <code>fstream</code> or <code>iostream</code> |
| <code>Stream_Var.get<br/>(Char_Variable);</code>                         | Reads one character from the input stream <i>Stream_Var</i> and sets the <i>Char_Variable</i> equal to this character. Does <i>not</i> skip over whitespace.                                                                                                                                                        | <code>fstream</code> or <code>iostream</code> |
| <code>Stream_Var.getline<br/>(String_Var,<br/>Max_Characters +1);</code> | One line of input from the stream <i>Stream_Var</i> is read, and the resulting string is placed in <i>String_Var</i> . If the line is more than <i>Max_Characters</i> long, only the first <i>Max_Characters</i> are read. The declared size of the <i>String_Var</i> should be <i>Max_Characters</i> +1 or larger. | <code>fstream</code> or <code>iostream</code> |
| <code>Stream_Var.peek( );</code>                                         | Reads one character from the input stream <i>Stream_Var</i> and returns that character. But the character read is <i>not</i> removed from the input stream; the next read will read the same character.                                                                                                             | <code>fstream</code> or <code>iostream</code> |

## Input and Output Member Functions (*continued*)

| Form of a Function Call                            | Description                                                                                                                                                                                 | Header File                          |
|----------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------|
| <i>Stream_Var.put</i><br>( <i>Char_Exp</i> );      | Writes the value of the <i>Char_Exp</i> to the output stream <i>Stream_Var</i> .                                                                                                            | <i>fstream</i> or<br><i>iostream</i> |
| <i>Stream_Var.putback</i><br>( <i>Char_Exp</i> );  | Places the value of <i>Char_Exp</i> in the input stream <i>Stream_Var</i> so that that value is the next input value read from the stream. The file connected to the stream is not changed. | <i>fstream</i> or<br><i>iostream</i> |
| <i>Stream_Var.precision</i><br>( <i>Int_Exp</i> ); | Specifies the number of digits output after the decimal point for floating-point values sent to the output stream <i>Stream_Var</i> .                                                       | <i>fstream</i> or<br><i>iostream</i> |
| <i>Stream_Var.width</i><br>( <i>Int_Exp</i> );     | Sets the field width for the next value output to the stream <i>Stream_Var</i> .                                                                                                            | <i>fstream</i> or<br><i>iostream</i> |
| <i>Stream_Var.setf</i> ( <i>Flag</i> );            | Sets flags for formatting output to the stream <i>Stream_Var</i> . See Display 6.5 for the list of possible flags.                                                                          | <i>fstream</i> or<br><i>iostream</i> |
| <i>Stream_Var.unsetf</i> ( <i>Flag</i> );          | Unsets flags for formatting output to the stream <i>Stream_Var</i> . See Display 6.5 for the list of possible flags.                                                                        | <i>fstream</i> or<br><i>iostream</i> |

## Character Functions

For all of these the actual type of the argument is *int*, but for most purposes you can think of the argument type as *char*. If the value returned is a value of type *int*, you must perform an explicit or implicit typecast to obtain a *char*.

| Function Declaration             | Description                                                                                                                                                                                                                | Header File |
|----------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------|
| <code>bool isalnum(char);</code> | Returns <i>true</i> if its argument satisfies either <code>isalpha</code> or <code>isdigit</code> . Otherwise, returns <i>false</i> .                                                                                      | cctype      |
| <code>bool isalpha(char);</code> | Returns <i>true</i> if its argument is an upper- or lowercase letter. It may also return <i>true</i> for other arguments. The details are implementation dependent. Otherwise, returns <i>false</i> .                      | cctype      |
| <code>bool isdigit(char);</code> | Returns <i>true</i> if its argument is a digit. Otherwise, returns <i>false</i> .                                                                                                                                          | cctype      |
| <code>bool ispunct(char);</code> | Returns <i>true</i> if its argument is a printable character that does not satisfy <code>isalnum</code> and is not whitespace. (These characters are considered punctuation characters.) Otherwise, returns <i>false</i> . | cctype      |
| <code>bool isspace(char);</code> | Returns <i>true</i> if its argument is a whitespace character (such as blank, tab, or new line). Otherwise, returns <i>false</i> .                                                                                         | cctype      |
| <code>bool iscntrl(char);</code> | Returns <i>true</i> if its argument is a control character. Otherwise, returns <i>false</i> .                                                                                                                              | cctype      |
| <code>bool islower(char);</code> | Returns <i>true</i> if its argument is a lowercase letter. Otherwise, returns <i>false</i> .                                                                                                                               | cctype      |
| <code>bool isupper(char);</code> | Returns <i>true</i> if its argument is an uppercase letter. Otherwise, returns <i>false</i> .                                                                                                                              | cctype      |
| <code>int tolower(char);</code>  | Returns the lowercase version of its argument. If there is no lowercase version, returns its argument unchanged.                                                                                                           | cctype      |
| <code>int toupper(char);</code>  | Returns the uppercase version of its argument. If there is no uppercase version, returns its argument unchanged.                                                                                                           | cctype      |

## String Functions

| Function Declaration                                             | Description                                                                                                                                                       | Header File          |
|------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------|
| <code>int atoi(const char a[]);</code>                           | Converts a string of characters to an integer.                                                                                                                    | cstdlib              |
| <code>int stoi(const string)</code>                              | Converts a STL string object to an integer. C++11 and higher.                                                                                                     | string               |
| <code>long atol(const char a[]);</code>                          | Converts a string of characters to a long integer.                                                                                                                | cstdlib              |
| <code>long stol(const string)</code>                             | Converts a STL string object to a long. C++11 and higher.                                                                                                         | string               |
| <code>double atof(const char a[]);</code>                        | Converts a string of characters to a double.                                                                                                                      | cstdlib <sup>1</sup> |
| <code>strcat(String_Variable, String_Expression);</code>         | Appends the value of the <i>String Expression</i> to the end of the string in the <i>String_Variable</i> .                                                        | cstring              |
| <code>strcmp(String_Exp1, String_Exp2)</code>                    | Returns <i>true</i> if the values of the two string expressions are different; otherwise, returns <i>false</i> . <sup>2</sup>                                     | cstring              |
| <code>strcpy(String_Variable, String_Expression);</code>         | Changes the value of the <i>String_Variable</i> to the value of the <i>String_Expression</i> .                                                                    | cstring              |
| <code>strlen(String_Expression)</code>                           | Returns the length of the <i>String_Expression</i> .                                                                                                              | cstring              |
| <code>strncat(String_Variable, String_Expression, Limit);</code> | Same as <code>strcat</code> except that at most <i>Limit</i> characters are appended.                                                                             | cstring              |
| <code>strncmp(String_Exp1, String_Exp2, Limit)</code>            | Same as <code>strcmp</code> except that at most <i>Limit</i> characters are compared.                                                                             | cstring              |
| <code>strncpy(String_Variable, String_Expression, Limit);</code> | Same as <code>strcpy</code> except that at most <i>Limit</i> characters are copied.                                                                               | cstring              |
| <code>strstr(String_Expression, Pattern)</code>                  | Returns a pointer to the first occurrence of the string <i>Pattern</i> in <i>String_Expression</i> . Returns the NULL pointer if the <i>Pattern</i> is not found. | cstring              |
| <code>strchr(String_Expression, Character)</code>                | Returns a pointer to the first occurrence of the <i>Character</i> in <i>String_Expression</i> . Returns the NULL pointer if <i>Character</i> is not found.        | cstring              |
| <code> strrchr(String_Expression, Character)</code>              | Returns a pointer to the last occurrence of the <i>Character</i> in <i>String_Expression</i> . Returns the NULL pointer if <i>Character</i> is not found.         | cstring              |

<sup>1</sup> Some implementations place it in `cmath`.

<sup>2</sup> Returns an integer that is less than zero, zero, or greater than zero according to whether *String\_Exp1* is less than, equal to, or greater than *String\_Exp2*, respectively. The ordering is lexicographic ordering.

## Random Number Generator

| Function Declaration                                                                                                                                                                                                                                                                            | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       | Header File |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------|
| <code>int random(int);</code>                                                                                                                                                                                                                                                                   | The call <code>random(n)</code> returns a pseudorandom integer greater than or equal to 0 and less than or equal to <code>n-1</code> . (Not available in all implementations. If not available, then you must use <code>rand</code> .)                                                                                                                                                                                                                                                            | cstdlib     |
| <code>int rand();</code>                                                                                                                                                                                                                                                                        | The call <code>rand()</code> returns a pseudorandom integer greater than or equal to 0 and less than or equal to <code>RAND_MAX</code> . <code>RAND_MAX</code> is a predefined integer constant that is defined in <code>cstdlib</code> . The value of <code>RAND_MAX</code> is implementation dependent but will be at least 32767.                                                                                                                                                              | cstdlib     |
| <code>void srand(unsigned int);</code><br><code>void srandom(unsigned int);</code><br><br>(The type <code>unsigned int</code> is an integer type that only allows nonnegative values. You can think of the argument type as <code>int</code> with the restriction that it must be nonnegative.) | Reinitializes the random number generator. The argument is the seed. Calling <code>srand</code> multiple times with the same argument will cause <code>rand</code> or <code>random</code> (whichever you use) to produce the same sequence of pseudorandom numbers. If <code>rand</code> or <code>random</code> is called without any previous call to <code>srand</code> , the sequence of numbers produced is the same as if there had been a call to <code>srand</code> with an argument of 1. | cstdlib     |

## Trigonometric Functions

These functions use radians, not degrees.

| Function Declaration              | Description        | Header File |
|-----------------------------------|--------------------|-------------|
| <code>double acos(double);</code> | Arc cosine         | cmath       |
| <code>double asin(double);</code> | Arc sine           | cmath       |
| <code>double atan(double);</code> | Arc tangent        | cmath       |
| <code>double cos(double);</code>  | Cosine             | cmath       |
| <code>double cosh(double);</code> | Hyperbolic cosine  | cmath       |
| <code>double sin(double);</code>  | Sine               | cmath       |
| <code>double sinh(double);</code> | Hyperbolic sine    | cmath       |
| <code>double tan(double);</code>  | Tangent            | cmath       |
| <code>double tanh(double);</code> | Hyperbolic tangent | cmath       |