CHAPTER 7

CLASSES

CONTENTS

In C++ we use classes to define our own data types. By defining types that mirror concepts in the problems we are trying to solve, we can make our programs easier to write, debug, and modify.

This chapter continues the coverage of classes begun in Chapter 2. Here we will focus on the importance of data abstraction, which lets us separate the implementation of an object from the operations that that object can perform. In Chapter 13 we'll learn how to control what happens when objects are copied, moved, assigned, or destroyed. In Chapter 14 we'll learn how to define our own operators.

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.

Note

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 isbns
            total.combine(trans);   // update the running total
        else {
            print(cout, total) << endl;   // print the results
            total = trans;                // process the next book
        }
    }
    print(cout, total) << endl;           // print the last transaction
} else {                                  // there was no input
    cerr << "No data?!" << endl;          // notify the user
}
```

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. Nonmember 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&);
```

Note: 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 non`const` 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.

> **Note:** 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.

Note

> 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 "`const`ness" 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.

Note: 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.

WARNING: 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.

C++ 11

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.

WARNING

> 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.

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 `vector`s and `string`s 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 `string`s.

WARNING

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`.

Note: 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).

Tip: 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.

Note: **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`.

Note: 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 `friend`s. 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_mgr`.

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.

> Note: 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_mgr` 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_mgr {
private:
    // Screens this Window_mgr is tracking
    // by default, a Window_mgr 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`).

Note

> 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.

> Note: 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 non`const` version of `display` calls `do_display`, its `this` pointer is implicitly converted from a pointer to non`const` 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 non`const` version, `this` points to a non`const` object, so that version of `display` returns an ordinary (non`const`) 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
```

> Note: 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 non`public` 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);
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`.

Note: 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_mgr` in which `clear` is a member of `Window_mgr` 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_mgr` (§ 7.3.4, p. 280). That function's parameter uses a type that is defined by `Window_mgr`:

```
void Window_mgr::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_mgr`, 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.

> **Note** 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.

> **Tip:** 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
}
```

Note: 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
}
```

Note: 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` of `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) { }
```

Note

> 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.

Best Practices

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.

Note: 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. C++ 11

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 expressions 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;
```

WARNING

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 execercise to hold objects of type `C`?

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

(a) A class must provide at least one constructor.
(b) A default constructor is a constructor with an empty parameter list.
(c) If there are no meaningful default values for a class, the class should not provide a default constructor.
(d) If a class does not define a default constructor, the compiler generates one 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.

> Note: 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;
```

Note

When a constructor is declared explicit, it can be used only with the direct form of initialization (§ 3.2.1, p. 84). Moroever, 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`: C++ 11

```
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) { }
```

```
        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 non`static` 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;
}
```

> **Note:** 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.

> **Tip:** 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 `constexpr`s 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
```

Best Practices 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 non`static` 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 non`static` 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 non`static` 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 non`static` 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;
```

Chapter Summary

Classes are the most fundamental feature in C++. Classes let us define new types for our applications, making our programs shorter and easier to modify.

Data abstraction—the ability to define both data and function members—and encapsulation—the ability to protect class members from general access—are fundamental to classes. We encapsulate a class by defining its implementation members as `private`. Classes may grant access to their non`public` member by designating another class or function as a friend.

Classes may define constructors, which are special member functions that control how objects are initialized. Constructors may be overloaded. Constructors should use a constructor initializer list to initialize all the data members.

Classes may also define `mutable` or `static` members. A `mutable` member is a data member that is never `const`; its value may be changed inside a `const` member function. A `static` member can be either function or data; `static` members exist independently of the objects of the class type.

Defined Terms

abstract data type Data structure that encapsulates (hides) its implementation.

access specifier Keywords `public` and `private`. Used to define whether members are accessible to users of the class or only to friends and members of the class. Specifiers may appear multiple times within a class. Each specifier sets the access of the following members up to the next specifier.

aggregate class Class with only `public` data members that has no in-class initializers or constructors. Members of an aggregate can be initialized by a brace-enclosed list of initializers.

class C++ mechanism for defining our own abstract data types. Classes may have data, function, or type members. A class defines a new type and a new scope.

class declaration The keyword `class` (or `struct`) followed by the class name followed by a semicolon. If a class is declared but not defined, it is an incomplete type.

class keyword Keyword used to define a class; by default members are `private`.

class scope Each class defines a scope. Class scopes are more complicated than other scopes—member functions defined within the class body may use names that appear even after the definition.

const member function A member function that may not change an object's ordinary (i.e., neither `static` nor `mutable`) data members. The `this` pointer in a `const` member is a pointer to `const`. A member function may be overloaded based on whether the function is `const`.

constructor A special member function used to initialize objects. Each constructor should give each data member a well-defined initial value.

constructor initializer list Specifies initial values of the data members of a class. The members are initialized to the values specified in the initializer list before the body of the constructor executes. Class members that are not initialized in the initializer list are default initialized.

converting constructor A non`explicit` constructor that can be called with a single argument. Such constructors implicitly convert from the argument's type to the class type.

data abstraction Programming technique that focuses on the interface to a type. Data abstraction lets programmers ignore the details of how a type is represented and think instead about the operations that the type can perform. Data abstraction is fundamental to both object-oriented and generic programming.

default constructor Constructor that is used if no initializer is supplied.

delegating constructor Constructor with a constructor-initializer list that has one entry that designates another constructor of the same class to do the initialization.

encapsulation Separation of implementation from interface; encapsulation hides the implementation details of a type. In C++, encapsulation is enforced by putting the implementation in the `private` part of a class.

explicit constructor Constructor that can be called with a single argument but cannot be used in an implicit conversion. A constructor is made explicit by prepending the keyword `explicit` to its declaration.

forward declaration Declaration of an as yet undefined name. Most often used to refer to the declaration of a class that appears prior to the definition of that class. See incomplete type.

friend Mechanism by which a class grants access to its non`public` members. Friends have the same access rights as members. Both classes and functions may be named as friends.

implementation The (usually `private`) members of a class that define the data and any operations that are not intended for use by code that uses the type.

incomplete type Type that is declared but not defined. It is not possible to use an incomplete type to define a variable or class member. It is legal to define references or pointers to incomplete types.

interface The (`public`) operations supported by a type. Ordinarily, the interface does not include data members.

member function Class member that is a function. Ordinary member functions are bound to an object of the class type through the implicit `this` pointer. `static` member functions are not bound to an object and have no `this` pointer. Member functions may be overloaded; when they are, the implicit `this` pointer participates in the function matching.

mutable data member Data member that is never `const`, even when it is a member of a `const` object. A `mutable` member can be changed inside a `const` function.

name lookup Process by which the use of a name is matched to its declaration.

private members Members defined after a `private` access specifier; accessible only to the friends and other class members. Data members and utility functions used by the class that are not part of the type's interface are usually declared `private`.

public members Members defined after a `public` access specifier; accessible to any user of the class. Ordinarily, only the functions that define the interface to the class should be defined in the `public` sections.

struct keyword Keyword used to define a class; by default members are `public`.

synthesized default constructor The default constructor created (synthesized) by the compiler for classes that do not explicitly define any constructors. This constructor initializes the data members from their in-class initializers, if present; otherwise it default initializes the data members.

this pointer Implicit value passed as an extra argument to every non`static` member function. The `this` pointer points to the object on which the function is invoked.

= default Syntax used after the parameter list of the declaration of the default constructor inside a class to signal to the compiler that it should generate the constructor, even if the class has other constructors.