Chapter 15: Object-Oriented Programming

From Lippman - Primer

Introduction

CONTENTS

Section 15.1 OOP: An Overview

Section 15.2 Defining Base and Derived Classes
Section 15.3 Conversions and Inheritance

Section 15.4 Constructors and Copy Control

Section 15.5 Class Scope under Inheritance

Section 15.6 Pure Virtual Functions

Section 15.7 Containers and Inheritance

Section 15.8 Handle Classes and Inheritance

Section 15.9 Text Queries Revisited

Chapter Summary

Defined Terms

Object-oriented programming is based on three fundamental concepts: data abstraction, inheritance, and dynamic binding. In C++ we use classes for data abstraction and class derivation to inherit one class from another: A derived class inherits the members of its base class(es). Dynamic binding lets the compiler determine at run time whether to use a function defined in the base or derived class.

Inheritance and dynamic binding streamline our programs in two ways: They make it easier to define new classes that are similar, but not identical, to other classes, and they make it easier for us to write programs that can ignore the details of how those similar types differ.

Many applications are characterized by concepts that are related but slightly different. For example, our bookstore might offer different pricing strategies for different books. Some books might be sold only at a given price. Others might be sold subject to some kind of discount strategy. We might give a discount to purchasers who buy a specified
number of copies of the book. Or we might give a discount for only the first few copies purchased but charge full price for any bought beyond a given limit.

Object-oriented programming (OOP) is a good match to this kind of application. Through inheritance we can define types that model the different kinds of books. Through dynamic binding we can write applications that use these types but that can ignore the type-dependent differences.

The ideas of inheritance and dynamic binding are conceptually simple but have profound implications for how we build our applications and for the features that programming languages must support. Before covering how C++ supports OOP, we’ll look at the concepts that are fundamental to this style of programming.

Section 15.1: OOP: An Overview

The key idea behind OOP is polymorphism. Polymorphism is derived from a Greek word meaning “many forms.” We speak of types related by inheritance as polymorphic types, because in many cases we can use the “many forms” of a derived or base type interchangeably. As we’ll see, in C++, polymorphism applies only to references or pointers to types related by inheritance.

Section : Inheritance

Inheritance lets us define classes that model relationships among types, sharing what is common and specializing only that which is inherently different. Members defined by the base class are inherited by its derived classes. The derived class can use, without change, those operations that do not depend on the specifics of the derived type. It can redefine those member functions that do depend on its type, specializing the function to take into account the peculiarities of the derived type. Finally, a derived class may define additional members beyond those it inherits from its base class.

Classes related by inheritance are often described as forming an inheritance hierarchy. There is one class, referred to as the root, from which all the other classes inherit, directly or indirectly. In our bookstore example, we will define a base class, which we’ll name Item_base, to represent undiscounted books. From Item_base we will inherit a second class, which we’ll name Bulk_item, to represent books sold with a quantity discount. At a minimum, these classes will define the following operations:

- an operation named book that will return the ISBN
- an operation named net_price that returns the price for purchasing a specified number of copies of a book

Classes derived from Item_base will inherit the book function without change: The derived classes have no need to redefine what it means to fetch the ISBN. On the other hand, each derived class will need to define its own version of the net_price function to implement an appropriate discount pricing strategy.

In C++, a base class must indicate which of its functions it intends for its derived classes to redefine. Functions defined as virtual are ones that the base expects its derived
classes to redefine. Functions that the base class intends its children to inherit are not defined as virtual.

Given this discussion, we can see that our classes will define three (const) member functions:

- A nonvirtual function, `std::string book()`, that returns the ISBN. It will be defined by `Item_base` and inherited by `Bulk_item`.
- Two versions of the virtual function, `double net_price(size_t)`, to return the total price for a given number of copies of a specific book. Both `Item_base` and `Bulk_item` will define their own versions of this function.

**Section : Dynamic Binding**

*Dynamic binding* lets us write programs that use objects of any type in an inheritance hierarchy without caring about the objects’ specific types. Programs that use these classes need not distinguish between functions defined in the base or in a derived class.

For example, our bookstore application would let a customer select several books in a single sale. When the customer was done shopping, the application would calculate the total due. One part of figuring the final bill would be to print for each book purchased a line reporting the total quantity and sales price for that portion of the purchase.

We might define a function named `print_total` to manage this part of the application. The `print_total` function, given an item and a count, should print the ISBN and the total price for purchasing the given number of copies of that particular book. The output of this function should look like:

```
ISBN: 0-201-54848-8 number sold: 3 total price: 98
ISBN: 0-201-82470-1 number sold: 5 total price: 202.5
```

Our `print_total` function might look something like the following:

```cpp
// calculate and print price for given number of copies, applying any discounts
void print_total(ostream &os, const Item_base &item, size_t n) {
        << "\ntnumber sold: " << n << "\ntotal price: "
        // virtual call: which version of net_price to call is resolved at run time
        << item.net_price(n) << endl;
}
```

The function’s work is trivial: It prints the results of calling `book` and `net_price` on its `item` parameter. There are two interesting things about this function.

First, even though its second parameter is a reference to `Item_base`, we can pass either an `Item_base` object or a `Bulk_item` object to this function.

Second, because the parameter is a reference and the `net_price` function is virtual, the call to `net_price` will be resolved at run time. The version of `net_price` that is called will depend on the type of the argument passed to `print_total`. When the argument to `print_total` is a `Bulk_item`, the version of `net_price` that is run will be the one defined in `Bulk_item` that applies a discount. If the argument is an `Item_base` object, then the call will be to the version defined by `Item_base`. 
In C++, dynamic binding happens when a virtual function is called through a reference (or a pointer) to a base class. The fact that a reference (or pointer) might refer to either a base- or a derived-class object is the key to dynamic binding. Calls to virtual functions made through a reference (or pointer) are resolved at run time: The function that is called is the one defined by the actual type of the object to which the reference (or pointer) refers.

Section 15.2: Defining Base and Derived Classes

In many ways, base and derived classes are defined like other classes we have already seen. However, there are some additional features that are required when defining classes in an inheritance hierarchy. This section will present those features. Subsequent sections will see how use of these features impacts classes and the programs we write using inherited classes.

Section 15.2.1: Defining a Base Class

Like any other class, a base class has data and function members that define its interface and implementation. In the case of our (very simplified) bookstore pricing application, our Item_base class defines the book and net_price functions and needs to store an ISBN and the standard price for the book:

```cpp
// Item sold at an undiscounted price
// derived classes will define various discount strategies
class Item_base {
public:
    Item_base(const std::string &book = "",
              double sales_price = 0.0):
        isbn(book), price(sales_price) { }
    std::string book() const { return isbn; }
    // returns total sales price for a specified number of items
    // derived classes will override and apply different discount algorithms
    virtual double net_price(std::size_t n) const
        { return n * price; }
    virtual ~Item_base() { }
private:
    std::string isbn;   // identifier for the item
    double price;       // normal, undiscounted price
};
```

For the most part, this class looks like others we have seen. It defines a constructor along with the functions we have already described. That constructor uses default arguments (Section 7.4.1, p. 253), which allows it to be called with zero, one, or two arguments. It initializes the data members from these arguments.
The new parts are the protected access label and the use of the virtual keyword on the destructor and the net_price function. We’ll explain virtual destructors in Section 15.4.4 (p. 587), but for now it is worth noting that classes used as the root class of an inheritance hierarchy generally define a virtual destructor.

Section : Base-Class Member Functions

The Item_base class defines two functions, one of which is preceded by the keyword virtual. The purpose of the virtual keyword is to enable dynamic binding. By default, member functions are nonvirtual. Calls to nonvirtual functions are resolved at compile time. To specify that a function is virtual, we precede its return type by the keyword virtual. Any nonstatic member function, other than a constructor, may be virtual. The virtual keyword appears only on the member-function declaration inside the class. The virtual keyword may not be used on a function definition that appears outside the class body.

We’ll have more to say about virtual functions in Section 15.2.4 (p. 566).

Best Practices

A base class usually should define as virtual any function that a derived class will need to redefine.

Section : Access Control and Inheritance

In a base class, the public and private labels have their ordinary meanings: User code may access the public members and may not access the private members of the class. The private members are accessible only to the members and friends of the base class. A derived class has the same access as any other part of the program to the public and private members of its base class: It may access the public members and has no access to the private members.

Sometimes a class used as a base class has members that it wants to allow its derived classes to access, while still prohibiting access to those same members by other users. The protected access label is used for such members. A protected member may be accessed by a derived object but may not be accessed by general users of the type.

Our Item_base class expects its derived classes to redefine the net_price function. To do so, those classes will need access to the price member. Derived classes are expected to access isbn in the same way as ordinary users: through the book access function.

Hence, the isbn member is private and is inaccessible to classes that inherit from Item_base.

Exercise 15.1:
What is a virtual member?
Exercise 15.2:
Define the protected access label. How does it differ from private?
Exercise 15.3:
Define your own version of the Item_base class.
Exercise 15.4:
A library has different kinds of materials that it lends out—books, CDs, DVDs, and so forth. Each of the different kinds of lending material has different check-in, check-out, and overdue rules. The following class defines a base class that we might use for this application. Identify which functions are likely to be defined as virtual and which, if any, are likely to be common among all lending materials. (Note: we assume that LibMember is a class representing a customer of the library, and Date is a class representing a calendar day of a particular year.)

```cpp
class Library {
public:
    bool check_out(const LibMember&);
    bool check_in (const LibMember&);
    bool is_late(const Date& today);
    double apply_fine();
    ostream& print(ostream& = cout);
    Date due_date() const;
    Date date_borrowed() const;
    string title() const;
    const LibMember& member() const;
};
```

Section 15.2.2: protected Members

The protected access label can be thought of as a blend of private and public:

- Like private members, protected members are inaccessible to users of the class.
- Like public members, the protected members are accessible to classes derived from this class.

In addition, protected has another important property:

- A derived object may access the protected members of its base class only through a derived object. The derived class has no special access to the protected members of base type objects.

As an example, let’s assume that Bulk_item defines a member function that takes a reference to a Bulk_item object and a reference to an Item_base object. This function may access the protected members of its own object as well as those of its Bulk_item parameter. However, it has no special access to the protected members in its Item_base parameter:

```cpp
void Bulk_item::memfcn(const Bulk_item &d, const Item_base &b)
{
    // attempt to use protected member
    double ret = price;      // ok: uses this->price
    ret = d.price;           // ok: uses price from a Bulk_item object
    ret = b.price;           // error: no access to price from an Item_base
}
```
The use of \texttt{d.price} is okay, because the reference to \texttt{price} is through an object of type \texttt{Bulk_item}. The use of \texttt{b.price} is illegal because \texttt{Bulk_item} has no special access to objects of type \texttt{Item_base}.

In the absence of inheritance, a class has two kinds of users: members of the class itself and the users of that class. This separation between kinds of users is reflected in the division of the class into \texttt{private} and \texttt{public} access levels. Users may access only the \texttt{public} interface; class members and friends may access both the \texttt{public} and \texttt{private} members.

Under inheritance, there is now a third kind of user of a class: programmers who will define new classes that are derived from the class. The provider of a derived class often (but not always) needs access to the (ordinarily \texttt{private}) base-class implementation. To allow that access while still preventing general access to the implementation, an additional access label, \texttt{protected}, is provided. The data and function members in a \texttt{protected} section of a class remain inaccessible to the general program, yet are accessible to the derived class. Anything placed within a \texttt{private} section of the base class is accessible only to the class itself and its friends. The \texttt{private} members are not accessible to the derived classes.

When designing a class to serve as a base class, the criteria for designating a member as \texttt{public} do not change: It is still the case that interface functions should be \texttt{public} and data generally should not be \texttt{public}. A class designed to be inherited from must decide which parts of the implementation to declare as \texttt{protected} and which should be \texttt{private}. A member should be made \texttt{private} if we wish to prevent subsequently derived classes from having access to that member. A member should be made \texttt{protected} if it provides an operation or data that a derived class will need to use in its implementation. In other words, the interface to the derived type is the combination of both the \texttt{protected} and \texttt{public} members.

**Section 15.2.3: Derived Classes**

To define a derived class, we use a \texttt{class derivation list} to specify the base class(es). A class derivation list names one or more base classes and has the form

\begin{verbatim}
  class classname: access-label base-class
\end{verbatim}

where \texttt{access-label} is one of \texttt{public}, \texttt{protected}, or \texttt{private}, and \texttt{base-class} is the name of a previously defined class. As we’ll see, a derivation list might name more than one base class. Inheritance from a single base class is most common and is the topic of this chapter. \texttt{Section 17.3} (p. 731) covers use of multiple base classes.

We’ll have more to say about the access label used in a derivation list in \texttt{Section 15.2.5} (p. 570). For now, what’s useful to know is that the access label determines the access to the inherited members. When we want to inherit the interface of a base class, then the derivation should be \texttt{public}.

A derived class inherits the members of its base class and may define additional members of its own. Each derived object contains two parts: those members that it inherits from its base and those it defines itself. Typically, a derived class \texttt{(re)}defines only those aspects that differ from or extend the behavior of the base.
Section : Defining a Derived Class

In our bookstore application, we will derive Bulk_item from Item_base, so Bulk_item will inherit the book, isbn, and price members. Bulk_item must redefine its net_price function and define the data members needed for that operation:

```cpp
class Bulk_item : public Item_base {
public:
    // redefines base version so as to implement bulk purchase discount policy
    double net_price(std::size_t) const;
private:
    std::size_t min_qty; // minimum purchase for discount to apply
    double discount; // fractional discount to apply
};
```

Each Bulk_item object contains four data elements: It inherits isbn and price from Item_base and defines min_qty and discount. These latter two members specify the minimum quantity and the discount to apply once that number of copies are purchased. The Bulk_item class also needs to define a constructor, which we shall do in Section 15.4 (p. 580).

Section : Derived Classes and virtual Functions

Ordinarily, derived classes redefine the virtual functions that they inherit, although they are not required to do so. If a derived class does not redefine a virtual, then the version it uses is the one defined in its base class.

A derived type must include a declaration for each inherited member it intends to redefine. Our Bulk_item class says that it will redefine the net_price function but will use the inherited version of book.

With one exception, the declaration (Section 7.4, p. 251) of a virtual function in the derived class must exactly match the way the function is defined in the base. That exception applies to virtuals that return a reference (or pointer) to a type that is itself a base class. A virtual function in a derived class can return a reference (or pointer) to a class that is publicly derived from the type returned by the base-class function.

For example, the Item_base class might define a virtual function that returned an Item_base*. If it did, then the instance defined in the Bulk_item class could be defined to return either an Item_base* or a Bulk_item*. We’ll see an example of this kind of virtual in Section 15.9 (p. 607).

Once a function is declared as virtual in a base class it remains virtual; nothing the derived classes do can change the fact that the function is virtual. When a derived class redefines a virtual, it may use the virtual keyword, but it is not required to do so.

Section : Derived Objects Contain Their Base Classes as Subobjects
A derived object consists of multiple parts: the (nonstatic) members defined in the derived class itself plus the subobjects made up of the (nonstatic) members of its base class. We can think of our `Bulk_item` class as consisting of two parts as represented in Figure 15.1.

There is no requirement that the compiler lay out the base and derived parts of an object contiguously. Hence, Figure 15.1 is a conceptual, not physical, representation of how classes work.

**Section: Functions in the Derived May Use Members from the Base**

As with any member function, a derived class function can be defined inside the class or outside, as we do here for the `net_price` function:

```cpp
// if specified number of items are purchased, use discounted price
double Bulk_item::net_price(size_t cnt) const
{
    if (cnt >= min_qty)
        return cnt * (1 - discount) * price;
    else
        return cnt * price;
}
```

This function generates a discounted price: If the given quantity is more than `min_qty`, we apply the `discount` (which was stored as a fraction) to the `price`.

Because each derived object has a base-class part, classes may access the public and protected members of its base class as if those members were members of the derived class itself.

**Section: A Class Must Be Defined to Be Used as a Base Class**

A class must be defined before it can be used as a base class. Had we declared, but not defined, `Item_base`, we could not use it as our base class:

```cpp
class Item_base; // declared but not defined
// error: Item_base must be defined
class Bulk_item : public Item_base { ... };
```
The reason for this restriction should already be easy to see: Each derived class contains, and may access, the members of its base class. To use those members, the derived class must know what they are. One implication of this rule is that it is impossible to derive a class from itself.

Section : Using a Derived Class as a Base Class

A base class can itself be a derived class:
```cpp
class Base { /* ... */ };
class D1: public Base { /* ... */ };
class D2: public D1 { /* ... */ };
```
Each class inherits all the members of its base class. The most derived type inherits the members of its base, which in turn inherits the members of its base and so on up the inheritance chain. Effectively, the most derived object contains a subobject for each of its immediate-base and indirect-base classes.

Section : Declarations of Derived Classes

If we need to declare (but not yet define) a derived class, the declaration contains the class name but does not include its derivation list. For example, the following forward declaration of `Bulk_item` results in a compile-time error:
```cpp
#error: a forward declaration must not include the derivation list
class Bulk_item : public Item_base;
```
The correct forward declarations are:
```cpp
// forward declarations of both derived and nonderived class
class Bulk_item;
class Item_base;
```

Section 15.2.4: virtual and Other Member Functions

By default, function calls in C++ do not use dynamic binding. To trigger dynamic binding, two conditions must be met: First, only member functions that are specified as virtual can be dynamically bound. By default, member functions are not virtual; nonvirtual functions are not dynamically bound. Second, the call must be made through a reference or a pointer to a base-class type. To understand this requirement, we need to understand what happens when we use a reference or pointer to an object that has a type from an inheritance hierarchy.

Exercise 15.5:
Which of the following declarations, if any, are incorrect?
```cpp
class Base { /* ... */ };
(a) class Derived : public Derived { /* ... */ };
(b) class Derived : Base { /* ... */ };
(c) class Derived : private Base { /* ... */ };
(d) class Derived : public Base;
(e) class Derived inherits Base { /* ... */ };
```

Exercise 15.6:
Write your own version of the `Bulk_item` class.

Exercise 15.7:
We might define a type to implement a limited discount strategy. This class would give a discount for books purchased up to a limit. If the number of copies purchased exceeds that limit, then the normal price should be applied to any books purchased beyond the limit. Define a class that implements this strategy.

**Section : Derived to Base Conversions**

Because every derived object contains a base part, we can bind a base-type reference to the base-class part of a derived object. We can also use a pointer to base to point to a derived object:

```cpp
// function with an Item_base reference parameter
double print_total(const Item_base&, size_t);
Item_base item;           // Object of base type
// ok: use pointer or reference to Item_base to refer to an
Item_base object
    print_total(item, 10);  // passes reference to an Item_base
object
    Item_base *p = &item;   // p points to an Item_base object
Bulk_item bulk;           // object of derived type
// ok: can bind a pointer or reference to Item_base to a Bulk_item
object
    print_total(bulk, 10);  // passes reference to the Item_base
part of bulk
    p = &bulk;              // p points to the Item_base part of
bulk
```

This code uses the same base-type pointer to point to an object of the base type and to an object of the derived type. It also calls a function that expects a reference to the base type, passing an object of the base-class type and also passing an object of the derived type. Both uses are fine, because every derived object has a base part.

Because we can use a base-type pointer or reference to refer to a derived-type object, when we use a base-type reference or pointer, we don’t know the type of the object to which the pointer or reference is bound: A base-type reference or pointer might refer to an object of base type or an object of derived type. Regardless of which actual type the object has, the compiler treats the object as if it is a base type object. Treating a derived object as if it were a base is safe, because every derived object has a base subobject. Also, the derived class inherits the operations of the base class, meaning that any operation that might be performed on a base object is available through the derived object as well.

The crucial point about references and pointers to base-class types is that the **static type**—the type of the reference or pointer, which is knowable at compile time—and the **dynamic type**—the type of the object to which the pointer or reference is bound, which is knowable only at run time—may differ.

**Section : Calls to virtual Functions May Be Resolved at Run time**
Binding a base-type reference or pointer to a derived object has no effect on the underlying object. The object itself is unchanged and remains a derived object. The fact that the actual type of the object might differ from the static type of the reference or pointer addressing that object is the key to dynamic binding in C++.

When a virtual function is called through a reference or pointer, the compiler generates code to decide at run time which function to call. The function that is called is the one that corresponds to the dynamic type. As an example, let’s look again at the `print_total` function:

```cpp
// calculate and print price for given number of copies, applying any discounts
void print_total(ostream &os,
    const Item_base &item, size_t n)
{
    << "\ntnumber sold: " << n << "\nttotal price: " // virtual call: which version of net_price to call is resolved at run time
    << item.net_price(n) << endl;
}
```

Because the `item` parameter is a reference and `net_price` is virtual, the version of `net_price` that is called in `item.net_price(n)` depends at run time on the actual type of the argument bound to the `item` parameter:

```cpp
Item_base base;
Bulk_item derived;
// print_total makes a virtual call to net_price
print_total(cout, base, 10); // calls Item_base::net_price
print_total(cout, derived, 10); // calls Bulk_item::net_price
```

In the first call, the `item` parameter is bound, at run time, to an object of type `Item_base`. As a result, the call to `net_price` inside `print_total` calls the version defined in `Item_base`. In the second call, `item` is bound to an object of type `Bulk_item`. In this call, the version of `net_price` called from `print_total` will be the one defined by the `Bulk_item` class.

The fact that the static and dynamic types of references and pointers can differ is the cornerstone of how C++ supports polymorphism.

When we call a function defined in the base class through a base-class reference or pointer, we do not know the precise type of the object on which the function is executed. The object on which the function executes might be of the base type or it might be an object of a derived type.

If the function called is nonvirtual, then regardless of the actual object type, the function that is executed is the one defined by the base type. If the function is virtual, then the decision as to which function to run is delayed until run time. The version of the virtual function that is run is the one defined by the type of the object to which the reference is bound or to which the pointer points.

From the perspective of the code that we write, we need not care. As long as the classes are designed and implemented correctly, the operations will do the right thing whether the actual object is of base or derived type.

On the other hand, an object is not polymorphic—its type is known and unchanging. The dynamic type of an object (as opposed to a reference or pointer) is always the same as the
static type of the object. The function that is run, virtual or nonvirtual, is the one defined by the type of the object.

Virtuals are resolved at run time only if the call is made through a reference or pointer. Only in these cases is it possible for an object’s dynamic type to be unknown until run time.

**Section: Nonvirtual Calls Are Resolved at Compile Time**

Regardless of the actual type of the argument passed to `print_total`, the call of `book` is resolved at compile time to `Item_base::book`.

Even if `Bulk_item` defined its own version of the `book` function, this call would call the one from the base class. Nonvirtual functions are always resolved at compile time based on the type of the object, reference, or pointer from which the function is called. The type of `item` is reference to `const Item_base`, so a call to a nonvirtual function on that object will call the one from `Item_base` regardless of the type of the actual object to which `item` refers at run time.

**Section: Overriding the Virtual Mechanism**

In some cases, we want to override the virtual mechanism and force a call to use a particular version of a virtual function. We can do so by using the scope operator:

```c++
Item_base *baseP = &derived;
// calls version from the base class regardless of the dynamic type of baseP
double d = baseP->Item_base::net_price(42);
```

This code forces the call to `net_price` to be resolved to the version defined in `Item_base`. The call will be resolved at compile time.

**Best Practices** Only code inside member functions should ever need to use the scope operator to override the virtual mechanism.

Why might we wish to override the virtual mechanism? The most common reason is when a derived-class virtual calls the version from the base. In such cases, the base-class version might do work common to all types in the hierarchy. Each derived type adds only whatever is particular to its own type.

For example, we might define a `Camera` hierarchy with a virtual `display` operation. The `display` function in the `Camera` class would display information common to all `Cameras`. A derived class, such as `PerspectiveCamera`, would need to display both that common information and the information unique to `PerspectiveCamera`. Rather than duplicate the `Camera` operations within `PerspectiveCamera`'s implementation of `display`, we could explicitly invoke the `Camera` version to display the common information. In a case
such as this one, we’d know exactly which instance to invoke, so there would be no need to go through the virtual mechanism.

When a derived virtual calls the base-class version, it must do so explicitly using the scope operator. If the derived function neglected to do so, then the call would be resolved at run time and would be a call to itself, resulting in an infinite recursion.

**Section : Virtual Functions and Default Arguments**

Like any other function, a virtual function can have default arguments. As usual, the value, if any, of a default argument used in a given call is determined at compile time. If a call omits an argument that has a default value, then the value that is used is the one defined by the type through which the function is called, irrespective of the object’s dynamic type. When a virtual is called through a reference or pointer to base, then the default argument is the value specified in the declaration of the virtual in the base class. If a virtual is called through a pointer or reference to derived, the default argument is the one declared in the version in the derived class.

Using different default arguments in the base and derived versions of the same virtual is almost guaranteed to cause trouble. Problems are likely to arise when the virtual is called through a reference or pointer to base, but the version that is executed is the one defined by the derived. In such cases, the default argument defined for the base version of the virtual will be passed to the derived version, which was defined using a different default argument.

**Section 15.2.5: Public, Private, and Protected Inheritance**

Access to members defined within a derived class is controlled in exactly the same way as access is handled for any other class ([Section 12.1.2, p. 432](#)). A derived class may define zero or more access labels that specify the access level of the members following that label. Access to the members the class inherits is controlled by a combination of the access level of the member in the base class and the access label used in the derived class’ derivation list.

**Exercise 15.8:**

Given the following classes, explain each print function:

```c++
struct base {
    string name() { return basename; }
    virtual void print(ostream &os) { os << basename; }
private:
    string basename;
};

struct derived {
```

```c++
```
Note: Each class controls access to the members it defines. A derived class may further restrict but may not loosen the access to the members that it inherits. The base class itself specifies the minimal access control for its own members. If a member is private in the base class, then only the base class and its friends may access that member. The derived class has no access to the private members of its base class, nor can it make those members accessible to its own users. If a base class member is public or protected, then the access label used in the derivation list determines the access level of that member in the derived class:

- **In public inheritance**, the members of the base retain their access levels: The public members of the base are public members of the derived and the protected members of the base are protected in the derived.
- **In protected inheritance**, the public and protected members of the base class are protected members in the derived class.
- **In private inheritance**, all the members of the base class are private in the derived class.

As an example, consider the following hierarchy:

```cpp
class Base {
public:
    void basemem();   // public member
protected:
    int i;            // protected member
    // ...
};
struct Public_derived : public Base {
    int use_base() { return i; } // ok: derived classes can access i
    // ...
};
struct Private_derived : private Base {
    int use_base() { return i; } // ok: derived classes can access i
};
```
All classes that inherit from Base have the same access to the members in Base, regardless of the access label in their derivation lists. The derivation access label controls the access that users of the derived class have to the members inherited from Base:

```cpp
Base b;
Public_derived d1;
Private_derived d2;
b.basemem();   // ok: basemem is public
d1.basemem();  // ok: basemem is public in the derived class
d2.basemem();  // error: basemem is private in the derived class
```

Both Public_derived and Private_derived inherit the basemem function. That member retains its access level when the inheritance is public, so d1 can call basemem. In Private_derived, the members of Base are private; users of Private_derived may not call basemem.

The derivation access label also controls access from indirectly derived classes:

```cpp
struct Derived_from_Private : public Private_derived {
    // error: Base::i is private in Private_derived
    int use_base() { return i; }
};

struct Derived_from_Public : public Public_derived {
    // ok: Base::i remains protected in PublicDerived
    int use_base() { return i; }
};
```

Classes derived from Public_derived may access i from the Base class because that member remains a protected member in Public_derived. Classes derived from Private_derived have no such access. To them all the members that Private_base inherited from Base are private.

### Section: Interface versus Implementation Inheritance

A publicly derived class inherits the interface of its base class; it has the same interface as its base class. In well-designed class hierarchies, objects of a publicly derived class can be used wherever an object of the base class is expected.

Classes derived using either private or protected do not inherit the base-class interface. Instead, these derivations are often referred to as implementation inheritance. The derived class uses the inherited class in its implementation but does not expose the fact of the inheritance as part of its interface.

As we’ll see in Section 15.3 (p. 577), whether a class uses interface or implementation inheritance has important implications for users of the derived class.

---

**Note**

By far the most common form of inheritance is public. The design of inheritance hierarchies is a complex topic in its own right and well beyond the scope of this language primer. However, there is one important design guide that is so fundamental that every programmer should be familiar with it.

When we define one class as publicly inherited from another, the derived class should reflect a so-called “Is A” relationship to the base class. In our bookstore example, our base class represents the concept of a book sold at a stipulated price. Our Bulk_item is a kind of book, but one with a different pricing strategy.
Another common relationship among types is a so-called “Has A” relationship. Our bookstore classes have a price and they have an ISBN. Types related by a “Has A” relationship imply membership. Thus, our bookstore classes are composed from members representing the price and the ISBN.

Section : Exempting Individual Members

When inheritance is private or protected, the access level of members of the base may be more restrictive in the derived class than it was in the base:

```cpp
class Base {
    public:
        std::size_t size() const { return n; }
    protected:
        std::size_t n;
};
class Derived : private Base { . . . };
```

The derived class can restore the access level of an inherited member. The access level cannot be made more or less restrictive than the level originally specified within the base class.

In this hierarchy, size is public in Base but private in Derived. To make size public in Derived we can add a using declaration for it to a public section in Derived.

```cpp
class Derived : private Base {
    public:
        // maintain access levels for members related to the size of the object
        using Base::size;
        protected:
        using Base::n;
        // ...
    };
```

Just as we can use a using declaration (Section 3.1, p. 78) to use names from the std namespace, we may also use a using declaration to access a name from a base class. The form is the same except that the left-hand side of the scope operator is a class name instead of a namespace name.

Section : Default Inheritance Protection Levels

In Section 2.8 (p. 65) we learned that classes defined with the struct and class keywords have different default access levels. Similarly, the default inheritance access level differs depending on which keyword is used to define the derived class. A derived class defined using the class keyword has private inheritance. A class is defined with the struct keyword, has public inheritance:

```cpp
class Base { /* ... */);
struct D1 : Base { /* ... */ };  // public inheritance by default
```
It is a common misconception to think that there are deeper differences between classes defined using the `struct` keyword and those defined using `class`. The only differences are the default protection level for members and the default protection level for a derivation. There are no other distinctions:

```cpp
class D3 : public Base {
public:
    /* ... */
};
// equivalent definition of D3
struct D3 : Base {
    // inheritance public by default
    /* ... */
};
struct D4 : private Base {
private:
    /* ... */
};
// equivalent definition of D4
class D4 : Base {
    // inheritance private by default
    /* ... */
};
```

### Best Practices

Although private inheritance is the default when using the `class` keyword, it is also relatively rare in practice. Because private inheritance is so rare, it is usually a good idea to explicitly specify `private`, rather than rely on the default. Being explicit makes it clear that private inheritance is intended and not an oversight.

**Exercise 15.10:**

In the exercises to Section 15.2.1 (p. 562) you wrote a base class to represent the lending policies of a library. Assume the library offers the following kinds of lending materials, each with its own check-out and check-in policy. Organize these items into an inheritance hierarchy:

- book
- audio book
- record
- children's puppet
- sega video game
- video
- cdrom book
- nintendo video game
- rental book
- sony play station
- video game

**Exercise 15.11:**

Choose one of the following general abstractions containing a family of types (or choose one of your own). Organize the types into an inheritance hierarchy:

- (a) Graphical file formats (such as gif, tiff, jpeg, bmp)
- (b) Geometric primitives (such as box, circle, sphere, cone)
- (c) C++ language types (such as class, function, member function)

**Exercise 15.12:**

For the class you chose in the previous exercise, identify some of the likely virtual functions as well as public and protected members.

### Section 15.2.6: Friendship and Inheritance
As with any other class, a base or derived class can make other class(es) or function(s) friends (Section 12.5, p. 465). Friends may access the class’ private and protected data.

Friendship is not inherited. Friends of the base have no special access to members of its derived classes. If a base class is granted friendship, only the base has special access. Classes derived from that base have no access to the class granting friendship. Each class controls friendship to its own members:

```cpp
class Base {
    friend class Frnd;
    protected:
        int i;
};
// Frnd has no access to members in D1
class D1 : public Base {
    protected:
        int j;
};
class Frnd {
    public:
        int mem(Base b) { return b.i; } // ok: Frnd is friend to Base
        int mem(D1 d) { return d.i; } // error: friendship doesn't inherit
    };
// D2 has no access to members in Base
class D2 : public Frnd {
    public:
        int mem(Base b) { return b.i; } // error: friendship doesn't inherit
    };
```

If a derived class wants to grant access to its members to the friends of its base class, the derived class must do so explicitly: Friends of the base have no special access to types derived from that base class. Similarly, if a base and its derived types all need access to another class, that class must specifically grant access to the base and each derived class.

### Section 15.2.7: Inheritance and Static Members

If a base class defines a static member (Section 12.6, p. 467) there is only one such member defined for the entire hierarchy. Regardless of the number of classes derived from the base class, there exists a single instance of each static member. static members obey normal access control: If the member is private in the base class, then derived classes have no access to it. Assuming the member is accessible, we can access the static member either through the base or derived class. As usual, we can use either the scope operator or the dot or arrow member access operators.

```cpp
struct Base {
    static void statmem(); // public by default
};
struct Derived : Base {
    void f(const Derived&); // public by default
};
```
void Derived::f(const Derived &derived_obj)
{
    Base::statmem();      // ok: Base defines statmem
    Derived::statmem();   // ok: Derived in herits statmem
    // ok: derived objects can be used to access static from base
    derived_obj.statmem(); // accessed through Derived object
    statmem();            // accessed through this class
}

Exercise 15.13:
Given the following classes, list all the ways a member function in C1 might access the static members of ConcreteBase. List all the ways an object of type C2 might access those members.

struct ConcreteBase {
    static std::size_t object_count();
    protected:
        static std::size_t obj_count;
};
struct C1 : public ConcreteBase { /* . . . */ };
struct C2 : public ConcreteBase { /* . . . */ };

Section 15.3: Conversions and Inheritance

Understanding conversions between base and derived types is essential to understanding how object-oriented programming works in C++. As we’ve seen, every derived object contains a base part, which means that we can execute operations on a derived object as if it were a base object. Because a derived object is also a base, there is an automatic conversion from a reference to a derived type to a reference to its base type(s). That is, we can convert a reference to a derived object to a reference to its base subobject and likewise for pointers.

Base-type objects can exist either as independent objects or as part of a derived object. Therefore, a base object might or might not be part of a derived object. As a result, there is no (automatic) conversion from reference (or pointer) to base to reference (or pointer) to derived.

The situation with respect to conversions of objects (as opposed to references or pointers) is more complicated. Although we can usually use an object of a derived type to initialize or assign an object of the base type, there is no direct conversion from an object of a derived type to an object of the base type.

Section 15.3.1: Derived-to-Base Conversions

If we have an object of a derived type, we can use its address to assign or initialize a pointer to the base type. Similarly, we can use a reference or object of the derived type to
initialize a reference to the base type. Pedantically speaking, there is no similar conversion for objects. The compiler will not automatically convert an object of derived type into an object of the base type. It is, however, usually possible to use a derived-type object to initialize or assign an object of base type. The difference between initializing and/or assigning an object and the automatic conversion that is possible for a reference or pointer is subtle and must be well understood.

Section : Conversion to a Reference is Not the Same as Converting an Object

As we’ve seen, we can pass an object of derived type to a function expecting a reference to base. We might therefore think that the object is converted. However, that is not what happens. When we pass an object to a function expecting a reference, the reference is bound directly to that object. Although it appears that we are passing an object, the argument is actually a reference to that object. The object itself is not copied and the conversion doesn’t change the derived-type object in any way. It remains a derived-type object.

When we pass a derived object to a function expecting a base-type object (as opposed to a reference) the situation is quite different. In that case, the parameter’s type is fixed—both at compile time and run time it will be a base-type object. If we call such a function with a derived-type object, then the base-class portion of that derived object is copied into the parameter.

It is important to understand the difference between converting a derived object to a base-type reference and using a derived object to initialize or assign to a base-type object.

Section : Using a Derived Object to Initialize or Assign a Base Object

When we initialize or assign an object of base type, we are actually calling a function: When we initialize, we’re calling a constructor; when we assign, we’re calling an assignment operator.

When we use a derived-type object to initialize or assign a base object, there are two possibilities. The first (albeit unlikely) possibility is that the base class might explicitly define what it means to copy or assign an object of the derived type to an object of the base type. It would do so by defining an appropriate constructor or assignment operator:

```cpp
class Derived;
class Base {
public:
    Base(const Derived&);  // create a new Base from a Derived
    Base &operator=(const Derived&);  // assign from a Derived
    // ...
};
```

In this case, the definition of these members would control what happens when a Derived object is used to initialize or assign to a Base object.

However, it is uncommon for classes to define explicitly how to initialize or assign an object of the base type from an object of derived type. Instead, base classes usually define (either explicitly or implicitly) their own copy constructor and assignment operator.
These members take a parameter that is a (const) reference to the base type. Because there is a conversion from reference to derived to reference to base, these copy-control members can be used to initialize or assign a base object from a derived object:

```cpp
Item_base item; // object of base type
Bulk_item bulk; // object of derived type
// ok: uses Item_base::Item_base(const Item_base&) constructor
Item_base item(bulk); // bulk is "sliced down" to its Item_base portion
// ok: calls Item_base::operator=(const Item_base&)
item = bulk; // bulk is "sliced down" to its Item_base portion
```

When we call the `Item_base` copy constructor or assignment operator on an object of type `Bulk_item`, the following steps happen:

1. The `Bulk_item` object is converted to a reference to `Item_base`, which means only that an `Item_base` reference is bound to the `Bulk_item` object.
2. That reference is passed as an argument to the copy constructor or assignment operator.
3. Those operators use the `Item_base` part of `Bulk_item` to initialize and assign, respectively, the members of the `Item_base` on which the constructor or assignment was called.
4. Once the operator completes, the object is an `Item_base`. It contains a copy of the `Item_base` part of the `Bulk_item` from which it was initialized or assigned, but the `Bulk_item` parts of the argument are ignored.

In these cases, we say that the `Bulk_item` portion of `bulk` is “sliced down” as part of the initialization or assignment to `item`. An `Item_base` object contains only the members defined in the base class. It does not contain the members defined by any of its derived types. There is no room in an `Item_base` object for the derived members.

**Section : Accessibility of Derived-to-Base Conversion**

Like an inherited member function, the conversion from derived to base may or may not be accessible. Whether the conversion is accessible depends on the access label specified on the derived class’ derivation.

To determine whether the conversion to base is accessible, consider whether a `public` member of the base class would be accessible. If so, the conversion is accessible; otherwise, it is not.

If the inheritance is `public`, then both user code and member functions of subsequently derived classes may use the derived-to-base conversion. If a class is derived using `private` or `protected` inheritance, then user code may not convert an object of derived type to a base type object. If the inheritance is `private`, then classes derived from the privately inherited class may not convert to the base class. If the inheritance is
protected, then the members of subsequently derived classes may convert to the base type.
Regardless of the derivation access label, a public member of the base class is accessible to the derived class itself. Therefore, the derived-to-base conversion is always accessible to the members and friends of the derived class itself.

Section 15.3.2: Conversions from Base to Derived

There is no automatic conversion from the base class to a derived class. We cannot use a base object when a derived object is required:

```c++
Item_base base;
Bulk_item* bulkP = &base;  // error: can't convert base to derived
Bulk_item& bulkRef = base; // error: can't convert base to derived
Bulk_item bulk = base;     // error: can't convert base to derived
```

The reason that there is no (automatic) conversion from base type to derived type is that a base object might be just that—a base. It does not contain the members of the derived type. If we were allowed to assign a base object to a derived type, then we might attempt to use that derived object to access members that do not exist.

What is sometimes a bit more surprising is that the restriction on converting from base to derived exists even when a base pointer or reference is actually bound to a derived object:

```c++
Bulk_item bulk;
Item_base *itemP = &bulk;  // ok: dynamic type is Bulk_item
Bulk_item *bulkP = itemP; // error: can't convert base to derived
```

The compiler has no way to know at compile time that a specific conversion will actually be safe at run time. The compiler looks only at the static types of the pointer or reference to determine whether a conversion is legal.

In those cases when we know that the conversion from base to derived is safe, we can use a static_cast (Section 5.12.4, p. 183) to override the compiler. Alternatively, we could request a conversion that is checked at run time by using a dynamic_cast, which is covered in Section 18.2.1 (p. 773).

Section 15.4: Constructors and Copy Control

The fact that each derived object consists of the (nonstatic) members defined in the derived class plus one or more base-class subobjects affects how derived-type objects are constructed, copied, assigned, and destroyed. When we construct, copy, assign, or destroy an object of derived type, we also construct, copy, assign, or destroy those base-class subobjects.

Constructors and the copy-control members are not inherited; each class defines its own constructor(s) and copy-control members. As is the case for any class, synthesized versions of the default constructor and the copy-control members will be used if the class does not define its own versions.
Section 15.4.1: Base-Class Constructors and Copy Control

Constructors and copy control for base classes that are not themselves a derived class are largely unaffected by inheritance. Our `Item_base` constructor looks like many we’ve seen before:

```cpp
Item_base(const std::string &book = ",",
          double sales_price = 0.0):
    isbn(book), price(sales_price) { }
```

The only impact inheritance has on base-class constructors is that there is a new kind of user that must be considered when deciding which constructors to offer. Like any other member, constructors can be made protected or private. Some classes need special constructors that are intended only for their derived classes to use. Such constructors should be made protected.

Section 15.4.2: Derived-Class Constructors

Derived constructors are affected by the fact that they inherit from another class. Each derived constructor initializes its base class in addition to initializing its own data members.

Section: The Synthesized Derived-Class Default Constructor

A derived-class synthesized default constructor ([Section 12.4.3](#), p. 458) differs from a nonderived constructor in only one way: In addition to initializing the data members of the derived class, it also initializes the base part of its object. The base part is initialized by the default constructor of the base class.

For our `Bulk_item` class, the synthesized default constructor would execute as follows:

1. Invoke the `Item_base` default constructor, which initializes the `isbn` member to the empty string and the `price` member to zero.
2. Initialize the members of `Bulk_item` using the normal variable initialization rules, which means that the `qty` and `discount` members would be uninitialized.

Section: Defining a Default Constructor

Because `Bulk_item` has members of built-in type, we should define our own default constructor:

```cpp
class Bulk_item : public Item_base {
public:
    Bulk_item(): min_qty(0), discount(0.0) { }
    // as before
};
```

This constructor uses the constructor initializer list ([Section 7.7.3](#), p. 263) to initialize its `min_qty` and `discount` members. The constructor initializer also implicitly invokes the `Item_base` default constructor to initialize its base-class part.
The effect of running this constructor is that first the Item_base part is initialized using the Item_base default constructor. That constructor sets isbn to the empty string and price to zero. After the Item_base constructor finishes, the members of the Bulk_item part are initialized, and the (empty) body of the constructor is executed.

**Section : Passing Arguments to a Base-Class Constructor**

In addition to the default constructor, our Item_base class lets users initialize the isbn and price members. We’d like to support the same initialization for Bulk_item objects. In fact, we’d like our users to be able to specify values for the entire Bulk_item, including the discount rate and quantity.

The constructor initializer list for a derived-class constructor may initialize only the members of the derived class; it may not directly initialize its inherited members. Instead, a derived constructor indirectly initializes the members it inherits by including its base class in its constructor initializer list:

```cpp
class Bulk_item : public Item_base {
public:
    Bulk_item(const std::string& book, double sales_price,
              std::size_t qty = 0, double disc_rate = 0.0):
        Item_base(book, sales_price),
        min_qty(qty), discount(disc_rate) { }

    // as before
};
```

This constructor uses the two-parameter Item_base constructor to initialize its base subobject. It passes its own book and sales_price arguments to that constructor. We might use this constructor as follows:

```cpp
// arguments are the isbn, price, minimum quantity, and discount
Bulk_item bulk("0-201-82470-1", 50, 5, .19);
```

bulk is built by first running the Item_base constructor, which initializes isbn and price from the arguments passed in the Bulk_item constructor initializer. After the Item_base constructor finishes, the members of Bulk_item are initialized. Finally, the (empty) body of the Bulk_item constructor is run.

---

The constructor initializer list supplies initial values for a class’ base class and members. It does not specify the order in which those initializations are done. The base class is initialized first and then the members of the derived class are initialized in the order in which they are declared.

**Section : Using Default Arguments in a Derived Constructor**

Of course, we might write these two Bulk_item constructors as a single constructor that takes default arguments:

```cpp
class Bulk_item : public Item_base {
public:
    Bulk_item(const std::string& book, double sales_price,
              std::size_t qty = 0, double disc_rate = 0.0):
        Item_base(book, sales_price),
        min_qty(qty), discount(disc_rate) { }
```
Here we provide defaults for each parameter so that the constructor might be used with zero to four arguments.

Section: Only an Immediate Base Class May Be Initialized

A class may initialize only its own immediate base class. An immediate base class is the class named in the derivation list. If class C is derived from class B, which is derived from class A, then B is the immediate base of C. Even though every C object contains an A part, the constructors for C may not initialize the A part directly. Instead, class C initializes B, and the constructor for class B in turn initializes A. The reason for this restriction is that the author of class B has specified how to construct and initialize objects of type B. As with any user of class B, the author of class C has no right to change that specification.

As a more concrete example, our bookstore might have several discount strategies. In addition to a bulk discount, it might offer a discount for purchases up to a certain quantity and then charge the full price thereafter. Or it might offer a discount for purchases above a certain limit but not for purchases up to that limit.

Each of these discount strategies is the same in that it requires a quantity and a discount amount. We might support these differing strategies by defining a new class named Disc_item to store the quantity and the discount amount. This class would not define a net_price function but would serve as a base class for classes such as Bulk_item that define the different discount strategies.

Adding Disc_item to the Item_base hierarchy is an example of refactoring. Refactoring involves redesigning a class hierarchy to move operations and/or data from one class to another. Refactoring happens most often when classes are redesigned to add new functionality or handle other changes in that application’s requirements.

Refactoring is common in OO applications. It is noteworthy that even though we changed the inheritance hierarchy, code that uses the Bulk_item or Item_base classes would not need to change. However, when classes are refactored, or changed in any other way, any code that uses those classes must be recompiled.

To implement this design, we first need to define the Disc_item class:

```cpp
// class to hold discount rate and quantity
// derived classes will implement pricing strategies using these data
class Disc_item : public Item_base {
  public:
    Disc_item(const std::string& book = "",
              double sales_price = 0.0,
              std::size_t qty = 0, double disc_rate = 0.0):
      Item_base(book, sales_price),
      quantity(qty), discount(disc_rate) { }

  protected:
    std::size_t quantity; // purchase size for discount to apply
    double discount;       // fractional discount to apply
};
```
This class inherits from `Item_base` and defines its own members, `discount` and `quantity`. Its only member function is the constructor, which initializes its `Item_base` base class and the members defined by `Disc_item`. Next, we can reimplement `Bulk_item` to inherit from `Disc_item`, rather than inheriting directly from `Item_base`:

```cpp
class Bulk_item : public Disc_item {
public:
    Bulk_item(const std::string& book = "",
              double sales_price = 0.0,
              std::size_t qty = 0, double disc_rate = 0.0):
        Disc_item(book, sales_price, qty, disc_rate) { }
    // redefines base version so as to implement bulk purchase
discount policy
    double net_price(std::size_t) const;
};
```

The `Bulk_item` class now has a `direct base class`, `Disc_item`, and an `indirect base class`, `Item_base`. Each `Bulk_item` object has three subobjects: an (empty) `Bulk_item` part and a `Disc_item` subobject, which in turn has an `Item_base` base subobject. Even though `Bulk_item` has no data members of its own, it defines a constructor in order to obtain values to use to initialize its inherited members. A derived constructor may initialize only its immediate base class. Naming `Item_base` in the `Bulk_item` constructor initializer would be an error. The reason that a constructor can initialize only its immediate base class is that each class defines its own interface. When we define `Disc_item`, we specify how to initialize a `Disc_item` by defining its constructors. Once a class has defined its interface, all interactions with objects of that class should be through that interface, even when those objects are part of a derived object. For similar reasons, derived-class constructors may not initialize and should not assign to the members of its base class. When those members are `public` or `protected`, a derived constructor could assign values to its base class members inside the constructor body. However, doing so would violate the interface of the base. Derived classes should respect the initialization intent of their base classes by using constructors rather than assigning to these members in the body of the constructor.

**Section 15.4.3: Copy Control and Inheritance**

Like any other class, a derived class may use the synthesized copy-control members described in Chapter 13. The synthesized operations copy, assign, or destroy the base-class part of the object along with the members of the derived part. The base part is copied, assigned, or destroyed by using the base class’ copy constructor, assignment operator, or destructor.

Exercise 15.14:
Redefine the `Bulk_item` and `Item_base` classes so that they each need to define only a single constructor.
Exercise 15.15:
Identify the base- and derived-class constructors for the library class hierarchy described in the first exercise on page 575.

Exercise 15.16:
Given the following base class definition,
```cpp
struct Base {
    Base(int val): id(val) { }
protected:
    int id;
};
```
explain why each of the following constructors is illegal.
(a) struct C1 : public Base {
    C1(int val): id(val) { }
};
(b) struct C2 : public C1 {
    C2(int val): Base(val), C1(val){ }
};
(c) struct C3 : public C1 {
    C3(int val): Base(val) { }
};
(d) struct C4 : public Base {
    C4(int val) : Base(id + val){ }
};
(e) struct C5 : public Base {
    C5() { }
};

Whether a class needs to define the copy-control members depends entirely on the class’ own direct members. A base class might define its own copy control while the derived uses the synthesized versions or vice versa.

Classes that contain only data members of class type or built-in types other than pointers usually can use the synthesized operations. No special control is required to copy, assign, or destroy such members. Classes with pointer members often need to define their own copy control to manage these members.

Our Item_base class and its derived classes can use the synthesized versions of the copy-control operations. When a Bulk_item is copied, the (synthesized) copy constructor for Item_base is invoked to copy the isbn and price members. The isbn member is copied by using the string copy constructor; the price member is copied directly. Once the base part is copied, then the derived part is copied. Both members of Bulk_item are doubles, and these members are copied directly. The assignment operator and destructor are handled similarly.

Section : Defining a Derived Copy Constructor

Note:
If a derived class explicitly defines its own copy constructor or assignment operator, that definition completely overrides the defaults. The copy constructor and assignment operator for inherited classes are responsible for copying or assigning their base-class components as well as the members in the class itself.
If a derived class defines its own copy constructor, that copy constructor usually should explicitly use the base-class copy constructor to initialize the base part of the object:

```cpp
class Base { /* ... */ }
class Derived: public Base {
public:
    // Base::Base(const Base&) not invoked automatically
    Derived(const Derived& d):
        Base(d) /* other member initialization */ { /*... */ }
};
```

The initializer `Base(d)` converts (Section 15.3, p. 577) the derived object, `d`, to a reference to its base part and invokes the base-class copy constructor. Had the initializer for the base class been omitted,

```cpp
    // probably incorrect definition of the Derived copy constructor
    Derived(const Derived& d) /* derived member initizations */
        { /*... */ }
```

the effect would be to run the `Base` default constructor to initialize the base part of the object. Assuming that the initialization of the `Derived` members copied the corresponding elements from `d`, then the newly constructed object would be oddly configured: Its `Base` part would hold default values, while its `Derived` members would be copies of another object.

**Section : Derived-Class Assignment Operator**

As usual, the assignment operator is similar to the copy constructor: If the derived class defines its own assignment operator, then that operator must assign the base part explicitly:

```cpp
// Base::operator=(const Base&) not invoked automatically
Derived &Derived::operator=(const Derived &rhs)
{
    if (this != &rhs) {
        Base::operator=(rhs); // assigns the base part
        // do whatever needed to clean up the old value in the
        derived part
        // assign the members from the derived
    }
    return *this;
}
```

The assignment operator must, as always, guard against self-assignment. Assuming the left- and right-hand operands differ, then we call the `Base` class assignment operator to assign the base-class portion. That operator might be defined by the class or it might be the synthesized assignment operator. It doesn’t matter—we can call it directly. The base-class operator will free the old value in the base part of the left-hand operand and will assign the new values from `rhs`. Once that operator finishes, we continue doing whatever is needed to assign the members in the derived class.

**Section : Derived-Class Destructor**

The destructor works differently from the copy constructor and assignment operator: The derived destructor is never responsible for destroying the members of its base objects.
The compiler always implicitly invokes the destructor for the base part of a derived object. Each destructor does only what is necessary to clean up its own members:

```cpp
class Derived: public Base {
public:
    // Base::~Base invoked automatically
    ~Derived() { /* do what it takes to clean up derived members */ }
};
```

Objects are destroyed in the opposite order from which they are constructed: The derived destructor is run first, and then the base-class destructors are invoked, walking back up the inheritance hierarchy.

### Section 15.4.4: Virtual Destructors

The fact that destructors for the base parts are invoked automatically has an important consequence for the design of base classes. When we delete a pointer that points to a dynamically allocated object, the destructor is run to clean up the object before the memory for that object is freed. When dealing with objects in an inheritance hierarchy, it is possible that the static type of the pointer might differ from the dynamic type of the object that is being deleted. We might delete a pointer to the base type that actually points to a derived object.

If we delete a pointer to base, then the base-class destructor is run and the members of the base are cleaned up. If the object really is a derived type, then the behavior is undefined. To ensure that the proper destructor is run, the destructor must be virtual in the base class:

```cpp
class Item_base {
public:
    // no work, but virtual destructor needed
    // if base pointer that points to a derived object is ever deleted
    virtual ~Item_base() { }
};
```

If the destructor is virtual, then when it is invoked through a pointer, which destructor is run will vary depending on the type of the object to which the pointer points:

```cpp
Item_base *itemP = new Item_base; // same static and dynamic type
delete itemP; // ok: destructor for Item_base called
itemP = new Bulk_item; // ok: static and dynamic types differ
delete itemP; // ok: destructor for Bulk_item called
```

Like other virtual functions, the virtual nature of the destructor is inherited. Therefore, if the destructor in the root class of the hierarchy is virtual, then the derived destructors will be virtual as well. A derived destructor will be virtual whether the class explicitly defines its destructor or uses the synthesized destructor.

Destructors for base classes are an important exception to the Rule of Three (Section 13.3, p. 485). That rule says that if a class needs a destructor, then the class almost surely needs the other copy-control members. A base class almost always needs a destructor so that it can make the destructor virtual. If a base class has an empty destructor in order to make it virtual, then the fact that the class has a destructor is not an indication that the assignment operator or copy constructor is also needed.
Best Practices
The root class of an inheritance hierarchy should define a virtual destructor even if the destructor has no work to do.

Section: Constructors and Assignment Are Not Virtual

Of the copy-control members, only the destructor should be defined as virtual. Constructors cannot be defined as virtual. Constructors are run before the object is fully constructed. While the constructor is running, the object’s dynamic type is not complete. Although we can define a virtual operator= member function in the base class, doing so does not affect the assignment operators used in the derived classes. Each class has its own assignment operator. The assignment operator in a derived class has a parameter that has the same type as the class itself. That type must differ from the parameter type for the assignment operator in any other class in the hierarchy.

Making the assignment operator virtual is likely to be confusing because a virtual function must have the same parameter type in base and derived classes. The base-class assignment operator has a parameter that is a reference to its own class type. If that operator is virtual, then each class gets a virtual member that defines an operator= that takes a base object. But this operator is not the same as the assignment operator for the derived class.

Beware: Making the class assignment operator virtual is likely to be confusing and unlikely to be useful.

Exercise 15.17:
Describe the conditions under which a class should have a virtual destructor.

Exercise 15.18:
What operations must a virtual destructor perform?

Exercise 15.19:
What if anything is likely to be incorrect about this class definition?

```
class AbstractObject {
public:
    virtual void doit();
    // other members not including any of the copy-control
    functions
};
```

Exercise 15.20:
Recalling the exercise from Section 13.3 (p. 487) in which you wrote a class whose copy-control members printed a message, add print statements to the constructors of the Item_base and Bulk_item classes. Define the copy-control members to do the same job as the synthesized versions but that also print a message. Now write programs using objects and functions that use the Item_base types. In each case, predict what objects will be created and destroyed and compare your predictions with what your programs
generate. Continue experimenting until you can correctly predict which copy-control members are executed for a given bit of code.

Section 15.4.5: Virtuals in Constructors and Destructors

A derived object is constructed by first running a base-class constructor to initialize the base part of the object. While the base-class constructor is executing, the derived part of the object is uninitialized. In effect, the object is not yet a derived object. When a derived object is destroyed, its derived part is destroyed first, and then its base parts are destroyed in the reverse order of how they were constructed. In both cases, while a constructor or destructor is running, the object is incomplete. To accommodate this incompleteness, the compiler treats the object as if its type changes during construction or destruction. Inside a base-class constructor or destructor, a derived object is treated as if it were an object of the base type.
The type of an object during construction and destruction affects the binding of virtual functions.

If a virtual is called from inside a constructor or destructor, then the version that is run is the one defined for the type of the constructor or destructor itself. This binding applies to a virtual whether the virtual is called directly by the constructor (or destructor) or is called indirectly from a function that the constructor (or destructor) called.
To understand this behavior, consider what would happen if the derived-class version of a virtual function were called from a base-class constructor (or destructor). The derived version of the virtual probably accesses members of the derived object. After all, if the derived-class version didn’t need to use members from the derived object, the derived class could probably use the definition from the base class. However, the members of the derived part of the object aren’t initialized while the base constructor (or destructor) is running. In practice, if such access were allowed, the program would probably crash.

Section 15.5: Class Scope under Inheritance

Each class maintains its own scope (Section 12.3, p. 444) within which the names of its members are defined. Under inheritance, the scope of the derived class is nested within the scope of its base classes. If a name is unresolved within the scope of the derived class, the enclosing base-class scope(s) are searched for a definition of that name. It is this hierarchical nesting of class scopes under inheritance that allows the members of the base class to be accessed directly as if they are members of the derived class. When we write

```cpp
Bulk_item bulk;
cout << bulk.book();
```

the use of the name `book` is resolved as follows:
1. bulk is an object of the Bulk_item class. The Bulk_item class is searched for book. That name is not found.

2. Because Bulk_item is derived from Item_Base, the Item_Base class is searched next. The name book is found in the Item_base class. The reference is resolved successfully.

Section 15.5.1: Name Lookup Happens at Compile Time

The static type of an object, reference, or pointer determines the actions that the object can perform. Even when the static and dynamic types might differ, as can happen when a reference or pointer to a base type is used, the static type determines what members can be used. As an example, we might add a member to the Disc_item class that returns a pair holding the minimum (or maximum) quantity and the discounted price:

```cpp
class Disc_item : public Item_base {
public:
    std::pair<size_t, double> discount_policy() const
    { return std::make_pair(quantity, discount); }
    // other members as before
};
```

We can access discount_policy only through an object, pointer, or reference of type Disc_item or a class derived from Disc_item:

```cpp
Bulk_item bulk;
Bulk_item *bulkP = &bulk;  // ok: static and dynamic types are the same
Item_base *itemP = &bulk;  // ok: static and dynamic types differ
bulkP->discount_policy();  // ok: bulkP has type Bulk_item*
itemP->discount_policy();  // error: itemP has type Item_base*
```

The call through itemP is an error because a pointer (reference or object) to a base type can access only the base parts of an object and there is no discount_policy member defined in the base class.

Exercise 15.21:
Redefine your Item_base hierarchy to include a Disc_item class.

Exercise 15.22:
Redefine Bulk_item and the class you implemented in the exercises from Section 15.2.3 (p. 567) that represents a limited discount strategy to inherit from Disc_item.

Section 15.5.2: Name Collisions and Inheritance

Although a base-class member can be accessed directly as if it were a member of the derived class, the member retains its base-class membership. Normally we do not care which actual class contains the member. We usually need to care only when a base- and derived-class member share the same name.

```
struct Base {
```

A derived-class member with the same name as a member of the base class hides direct access to the base-class member.
struct Base {
    // Base(): mem(0) { }
    protected:
        int mem;
};
struct Derived : Base {
    Derived(int i): mem(i) { } // initializes Derived::mem
    int get_mem() { return mem; } // returns Derived::mem
    protected:
        int mem; // hides mem in the base
};

The reference to mem inside get_mem is resolved to the name inside Derived. Were we to write
    Derived d(42);
    cout << d.get_mem() << endl; // prints 42
then the output would be 42.

Section : Using the Scope Operator to Access Hidden Members

We can access a hidden base-class member by using the scope operator:
struct Derived : Base {
    int get_base_mem() { return Base::mem; }
};

The scope operator directs the compiler to look for mem starting in Base.

Best Practices
When designing a derived class, it is best to avoid name collisions with members of the base class whenever possible.

Exercise 15.23:
Given the following base- and derived-class definitions

struct Base {
    foo(int);
    protected:
        int bar;
        double foo_bar;
};

struct Derived : public Base {
    foo(string);
    bool bar(Base *pb);
    void foobar();
    protected:
        string bar;
};

identify the errors in each of the following examples and how each might be fixed:
(a) Derived d; d.foo(1024);
(b) void Derived::foobar() { bar = 1024; }
(c) bool Derived::bar(Base *pb)
    { return foo_bar == pb->foo_bar; }

Section 15.5.3: Scope and Member Functions
A member function with the same name in the base and derived class behaves the same way as a data member: The derived-class member hides the base-class member within the scope of the derived class. The base member is hidden, even if the prototypes of the functions differ:

```cpp
struct Base {
    int memfcn();
};
struct Derived : Base {
    int memfcn(int); // hides memfcn in the base
};
Derived d; Base b;
b.memfcn();        // calls Base::memfcn
d.memfcn(10);     // calls Derived::memfcn
d.memfcn();       // error: memfcn with no arguments is hidden
d.Base::memfcn(); // ok: calls Base::memfcn
```

The declaration of `memfcn` in `Derived` hides the declaration in `Base`. Not surprisingly, the first call through `b`, which is a `Base` object, calls the version in the base class. Similarly, the second call through `d` calls the one from `Derived`. What can be surprising is the third call:

```cpp
d.memfcn(); // error: Derived has no memfcn that takes no arguments
```

To resolve this call, the compiler looks for the name `memfcn`, which it finds in the class `Derived`. Once the name is found, the compiler looks no further. This call does not match the definition of `memfcn` in `Derived`, which expects an `int` argument. The call provides no such argument and so is in error.

Recall that functions declared in a local scope do not overload functions defined at global scope (Section 7.8.1, p. 268). Similarly, functions defined in a derived class do not overload members defined in the base. When the function is called through a derived object, the arguments must match a version of the function defined in the derived class. The base class functions are considered only if the derived does not define the function at all.

### Section : Overloaded Functions

As with any other function, a member function (virtual or otherwise) can be over-loaded. A derived class can redefine zero or more of the versions it inherits.

If the derived class redefines any of the overloaded members, then only the one(s) redefined in the derived class are accessible through the derived type. If a derived class wants to make all the overloaded versions available through its type, then it must either redefine all of them or none of them. Sometimes a class needs to redefine the behavior of only some of the versions in an overloaded set, and wants to inherit the meaning for others. It would be tedious in such
cases to have to redefine every base-class version in order to redefine the ones that the class needs to specialize.

Instead of redefining every base-class version that it inherits, a derived class can provide a using declaration (Section 15.2.5, p. 574) for the overloaded member. A using declaration specifies only a name; it may not specify a parameter list. Thus, a using declaration for a base-class member function name adds all the overloaded instances of that function to the scope of the derived-class. Having brought all the names into its scope, the derived class need redefine only those functions that it truly must define for its type. It can use the inherited definitions for the others.

Section 15.5.4: Virtual Functions and Scope

Recall that to obtain dynamic binding, we must call a virtual member through a reference or a pointer to a base class. When we do so, the compiler looks for the function in the base class. Assuming the name is found, the compiler checks that the arguments match the parameters.

We can now understand why virtual functions must have the same prototype in the base and derived classes. If the base member took different arguments than the derived-class member, there would be no way to call the derived function from a reference or pointer to the base type. Consider the following (artificial) collection of classes:

```c++
class Base {
public:
    virtual int fcn();
};
class D1 : public Base {
public:
    // hides fcn in the base; this fcn is not virtual
    int fcn(int); // parameter list differs from fcn in Base
    // D1 inherits definition of Base::fcn()
};
class D2 : public D1 {
public:
    int fcn(int); // nonvirtual function hides D1::fcn(int)
    int fcn();    // redefines virtual fcn from Base
};
```

The version of `fcn` in `D1` does not redefine the virtual `fcn` from `Base`. Instead, it hides `fcn` from the base. Effectively, `D1` has two functions named `fcn`: The class inherits a virtual named `fcn` from the `Base` and defines its own, nonvirtual member named `fcn` that takes an `int` parameter. However, the virtual from the `Base` cannot be called from a `D1` object (or reference or pointer to `D1`) because that function is hidden by the definition of `fcn(int)`.

The class `D2` redefines both functions that it inherits. It redefines the virtual version of `fcn` originally defined in `Base` and the nonvirtual defined in `D1`.

Section: Calling a Hidden Virtual through the Base Class

When we call a function through a base-type reference or pointer, the compiler looks for that function in the base class and ignores the derived classes:
All three pointers are pointers to the base type, so all three calls are resolved by looking in `Base` to see if `fcn` is defined. It is, so the calls are legal. Next, because `fcn` is virtual, the compiler generates code to make the call at run time based on the actual type of the object to which the reference or pointer is bound. In the case of `bp2`, the underlying object is a `D1`. That class did not redefine the virtual version of `fcn` that takes no arguments. The call through `bp2` is made (at run time) to the version defined in `Base`. Understanding how function calls are resolved is crucial to understanding inheritance hierarchies in C++. The following four steps are followed:

1. Start by determining the static type of the object, reference, or pointer through which the function is called.
2. Look for the function in that class. If it is not found, look in the immediate base class and continue up the chain of classes until either the function is found or the last class is searched. If the name is not found in the class or its enclosing base classes, then the call is in error.
3. Once the name is found, do normal type-checking ([Section 7.1.2](#), p. 229) to see if this call is legal given the definition that was found.
4. Assuming the call is legal, the compiler generates code. If the function is virtual and the call is through a reference or pointer, then the compiler generates code to determine which version to run based on the dynamic type of the object. Otherwise, the compiler generates code to call the function directly.

### Section 15.6: Pure Virtual Functions

The `Disc_item` class that we wrote on page 583 presents an interesting problem: That class inherits the `net_price` function from `Item_base` but does not redefine it. We didn’t redefine `net_price` because there is no meaning to ascribe to that function for the `Disc_item` class. A `Disc_item` doesn’t correspond to any discount strategy in our application. This class exists solely for other classes to inherit from it. We don’t intend for users to define `Disc_item` objects. Instead, `Disc_item` objects should exist only as part of an object of a type derived from `Disc_item`. However, as defined, there is nothing that prevents users from defining a plain `Disc_item` object. That leaves open the question of what would happen if a user did create a `Disc_item` and invoked `net_price` function on it. We now know from the scope discussion in the previous section that the effect would be to call the `net_price` function inherited from `Item_base`, which generates the undiscounted price.

**Exercise 15.24:**

Why is it that, given

```c++
Bulk_item bulk;
Item_base item(bulk);
Item_base *p = &bulk;
```
the expression
   p->net_price(10);
invokes the Bulk_item instance of net_price, whereas
   item.net_price(10);
invokes the Item_base instance?

Exercise 15.25:
Assume Derived inherits from Base and that Base defines each of the following
functions as virtual. Assuming Derived intends to define its own version of the virtual,
determine which declarations in Derived are in error and specify what’s wrong.
(a) Base* Base::copy(Base*);
   Derived* Derived::copy(Derived*);
(b) Base* Base::copy(Base*);
   Derived* Derived::copy(Base*);
(c) ostream& Base::print(int, ostream&=cout);
   ostream& Derived::print(int, ostream&);
(d) void Base::eval() const;
   void Derived::eval();

It’s hard to say what behavior users might expect from calling net_price on a
Disc_item. The real problem is that we’d rather they couldn’t create such objects at all.
We can enforce this design intent and correctly indicate that there is no meaning for the
Disc_item version of net_price by making net_price a pure virtual function. A pure
virtual function is specified by writing = 0 after the function parameter list:
   class Disc_item : public Item_base {
      public:
         double net_price(std::size_t) const = 0;
   };

Defining a virtual as pure indicates that the function provides an interface for sub-
sequent types to override but that the version in this class will never be called. As importantly,
users will not be able to create objects of type Disc_item.

An attempt to create an object of an abstract base class is a compile-time error:
   // Disc_item declares pure virtual functions
   Disc_item discounted; // error: can't define a Disc_item object
   Bulk_item bulk;       // ok: Disc_item subobject within Bulk_item

A class containing (or inheriting) one or more pure virtual functions is an abstract
base class. We may not create objects of an abstract type except as parts of objects of
classes derived from the abstract base.

Exercise 15.26:
Make your version of the Disc_item class an abstract class.

Exercise 15.27:
Try to define an object of type Disc_item and see what errors you get from the compiler.

Section 15.7: Containers and Inheritance

We’d like to use containers (or built-in arrays) to hold objects that are related by
inheritance. However, the fact that objects are not polymorphic (Section 15.3.1, p. 577)
affects how we can use containers with types in an inheritance hierarchy.
As an example, our bookstore application would probably have the notion of a basket that represents the books a customer is buying. We’d like to be able to store the purchases in a multiset (Section 10.5, p. 375). To define the multiset, we must specify the type of the objects that the container will hold. When we put an object in a container, the element is copied (Section 9.3.3, p. 318).

If we define the multiset to hold objects of the base type

```cpp
multiset<Item_base> basket;
Item_base base;
Bulk_item bulk;
basket.insert(base);  // ok: add copy of base to basket
basket.insert(bulk);  // ok: but bulk sliced down to its base part
```

then when we add objects that are of the derived type, only the base portion of the object is stored in the container. Remember, when we copy a derived object to a base object, the derived object is sliced down (Section 15.3.1, p. 577).

The elements in the container are Item_base objects. Regardless of whether the element was made as a copy of a Bulk_item object, when we calculate the net_price of an element the element would be priced without a discount. Once the object is put into the multiset, it is no longer a derived object.

Because derived objects are “sliced down” when assigned to a base object, containers and types related by inheritance do not mix well. We cannot fix this problem by defining the container to hold derived objects. In this case, we couldn’t put objects of Item_base into the container—there is no standard conversion from base to derived type. We could explicitly cast a base-type object into a derived and add the resulting object to the container. However, if we did so, disaster would strike when we tried to use such an element. In this case, the element would be treated as if it were a derived object, but the members of the derived part would be uninitialized.

The only viable alternative would be to use the container to hold pointers to our objects. This strategy works—but at the cost of pushing onto our users the problem of managing the objects and pointers. The user must ensure that the objects pointed to stay around for as long as the container. If the objects are dynamically allocated, then the user must ensure that they are properly freed when the container goes away. The next section presents a better and more common solution to this problem.

Exercise 15.28:
Define a vector to hold objects of type Item_base and copy a number of objects of type Bulk_item into the vector. Iterate over the vector and generate the net_price for the elements in the container.

Exercise 15.29:
Repeat your program, but this time store pointers to objects of type Item_base. Compare the resulting sum.

Exercise 15.30:
Explain any discrepancy in the amount generated by the previous two programs. If there is no discrepancy, explain why there isn’t one.
Section 15.8: Handle Classes and Inheritance

One of the ironies of object-oriented programming in C++ is that we cannot use objects to support it. Instead, we must use pointers and references, not objects. For example, in the following code fragment,

```cpp
void get_prices(Item_base object,
    const Item_base * pointer,
    const Item_base & reference)
{
    // which version of net_price is called is determined at run time
    cout << pointer->net_price(1) << endl;
    cout << reference.net_price(1) << endl;
    // always invokes Item_base::net_price
    cout << object.net_price(1) << endl;
}
```

the invocations through `pointer` and `reference` are resolved at run time based on the dynamic types of the object to which they are bound.

Unfortunately, using pointers or references puts a burden on the users of our classes. We saw one such burden in the previous section that discussed the interactions between objects of inherited types and containers.

A common technique in C++ is to define a so-called cover or handle class. The handle class stores and manages a pointer to the base class. The type of the object to which that pointer points will vary; it can point at either a base- or a derived-type object. Users access the operations of the inheritance hierarchy through the handle. Because the handle uses its pointer to execute those operations, the behavior of virtual members will vary at run time depending on the kind of object to which the handle is actually bound. Users of the handle thus obtain dynamic behavior but do not themselves have to worry about managing the pointer.

Handles that cover an inheritance hierarchy have two important design considerations:

- As with any class that holds a pointer (Section 13.5, p. 492), we must decide what to do about copy control. Handles that cover an inheritance hierarchy typically behave like either a smart pointer (Section 13.5.1, p. 495) or a value (Section 13.5.2, p. 499).
- The handle class determines whether the handle interface will hide the inheritance hierarchy or expose it. If the hierarchy is not hidden, users must know about and use objects in the underlying hierarchy.

There is no one right choice among these options; the decisions depend on the details of the hierarchy and how the class designer wants programmers to interact with those class(es). In the next two sections, we’ll implement two different kinds of handles that address these design questions in different ways.
Section 15.8.1: A Pointerlike Handle

As our first example, we’ll define a pointerlike handle class, named `Sales_item`, to represent our `Item_base` hierarchy. Users of `Sales_item` will use it as if it were a pointer: Users will bind a `Sales_item` to an object of type `Item_base` and will then use the * and -> operations to execute `Item_base` operations:

```cpp
// bind a handle to a Bulk_item object
Sales_item item(Bulk_item("0-201-82470-1", 35, 3, .20));
item->net_price(); // virtual call to net_price function
```

However, users won’t have to manage the object to which the handle points; the `Sales_item` class will do that part of the job. When users call a function through a `Sales_item`, they’ll get polymorphic behavior.

Section: Defining the Handle

We’ll give our class three constructors: a default constructor, a copy constructor, and a constructor that takes an `Item_base`. This third constructor will copy the `Item_base` and ensure that the copy stays around as long as the `Sales_item` does. When we copy or assign a `Sales_item`, we’ll copy the pointer rather than copying the object. As with our other pointerlike handle classes, we’ll use a use count to manage the copies.

The use-counted classes we’ve used so far have used a companion class to store the pointer and associated use count. In this class, we’ll use a different design, as illustrated in Figure 15.2. The `Sales_item` class will have two data members, both of which are pointers: One pointer will point to the `Item_base` object and the other will point to the use count. The `Item_base` pointer might point to an `Item_base` object or an object of a type derived from `Item_base`. By pointing to the use count, multiple `Sales_item` objects can share the same counter.

![Figure 15.2: Use-Count Strategy for the Sales_item Handle Class](image)

In addition to managing the use count, the `Sales_item` class will define the dereference and arrow operators:

```cpp
// use counted handle class for the Item_base hierarchy
class Sales_item {
public:
    // default constructor: unbound handle
    Sales_item(): p(0), use(new std::size_t(1)) { }
    // attaches a handle to a copy of the Item_base object
    Sales_item(const Item_base&);
```
// copy control members to manage the use count and pointers
Sales_item(const Sales_item &i):
    p(i.p), use(i.use) { ++*use; }
~Sales_item() { decr_use(); }
Sales_item& operator=(const Sales_item&);
// member access operators
const Item_base *operator->() const { if (p) return p;
    else throw std::logic_error("unbound Sales_item"); }
const Item_base &operator*() const { if (p) return *p;
    else throw std::logic_error("unbound Sales_item"); }

private:
    Item_base *p;        // pointer to shared item
    std::size_t *use;    // pointer to shared use count
    // called by both destructor and assignment operator to free
    // pointers
    void decr_use()
    { if (--*use == 0) { delete p; delete use; } }
};

Section : Use-Counted Copy Control

The copy-control members manipulate the use count and the Item_base pointer as appropriate. Copying a Sales_item involves copying the two pointers and incrementing the use count. The destructor decrements the use count and destroys the pointers if the count goes to zero. Because the assignment operator will need to do the same work, we implement the destructor’s actions in a private utility function named decr_use.

The assignment operator is a bit more complicated than the copy constructor:

// use-counted assignment operator; use is a pointer to a shared
use count
Sales_item&
Sales_item::operator=(const Sales_item &rhs)
{  
    ++*rhs.use;
    decr_use();
    p = rhs.p;
    use = rhs.use;
    return *this;
}

The assignment operator acts like the copy constructor in that it increments the use count of the right-hand operand and copies the pointer. It also acts like the destructor in that we first have to decrement the use count of the left-hand operand and then delete the pointers if the use count goes to zero.

As usual with an assignment operator, we must protect against self-assignment. This operator handles self-assignment by first incrementing the use count in the right-hand operand. If the left- and right-hand operands are the same, the use count will be at least 2 when decr_use is called. That function decrements and checks the use count of the left-hand operand. If the use count goes to zero, then decr_use will free the Item_base and use objects currently in this object. What remains is to copy the pointers from the right-hand to the left-hand operand. As usual, our assignment operator returns a reference to the left-hand operand.
Aside from the copy-control members, the only other functions Sales_item defines are the operator functions, operator* and operator->. Users will access Item_base members through these operators. Because these operators return a pointer and reference, respectively, functions called through these operators will be dynamically bound. We define only the const versions of these operators because the public members in the underlying Item_base hierarchy are all const.

Section : Constructing the Handle

Our handle has two constructors: the default constructor, which creates an un-bound Sales_item, and a second constructor, which takes an object to which it attaches the handle. The first constructor is easy: We set the Item_base pointer to 0 to indicate that this handle is not attached to any object. The constructor allocates a new use counter and initializes it to 1. The second constructor is more difficult. We’d like users of our handle to create their own objects, to which they could attach a handle. The constructor will allocate a new object of the appropriate type and copy the parameter into that newly allocated object. That way the Sales_item class will own the object and can guarantee that the object is not deleted until the last Sales_item attached to the object goes away.

Section 15.8.2: Cloning an Unknown Type

To implement the constructor that takes an Item_base, we must first solve a problem: We do not know the actual type of the object that the constructor is given. We know that it is an Item_base or an object of a type derived from Item_base. Handle classes often need to allocate a new copy of an existing object without knowing the precise type of the object. Our Sales_item constructor is a good example.

The common approach to solving this problem is to define a virtual operation to do the copy, which we’ll name clone. To support our handle class, we’ll need to add clone to each of the types in the hierarchy, starting with the base class, which must define the function as virtual:

```cpp
class Item_base {
    public:
        virtual Item_base* clone() const
        { return new Item_base(*this); }
};
```

Each class must now redefine the virtual. Because the function exists to generate a new copy of an object of the class, we’ll define the return type to reflect the type of the class itself:

```cpp
class Bulk_item : public Item_base {
    public:
        Bulk_item* clone() const
        { return new Bulk_item(*this); }
};
```
On page 564 we said there is one exception to the requirement that the return type of the derived class must match exactly that of the base class instance. That exception supports cases such as this one. If the base instance of a virtual function returns a reference or pointer to a class type, the derived version of the virtual may return a class publicly derived from the class returned by the base class instance (or a pointer or a reference to a class type).

Section : Defining the Handle Constructors

Once the clone function exists, we can write the Sales_item constructor:

```
Sales_item::Sales_item(const Item_base &item):
    p(item.clone()), use(new std::size_t(1)) { }
```

Like the default constructor, this constructor allocates and initializes its use count. It calls clone on its parameter to generate a (virtual) copy of that object. If the argument is an Item_base, then the clone function for Item_base is run; if the argument is a Bulk_item, then the Bulk_item clone is executed.

Exercise 15.31:
Define and implement the clone operation for the limited discount class implemented in the exercises for Section 15.2.3 (p. 567).

Exercise 15.32:
In practice, our programs are unlikely to run correctly the first time we run them or the first time we run them against real data. It is often useful to incorporate a debugging strategy into the design of our classes. Implement a virtual debug function for our Item_base class hierarchy that displays the data members of the respective classes.

Exercise 15.33:
Given the version of the Item_base hierarchy that includes the Disc_item abstract base class, indicate whether the Disc_item class should implement the clone function. If not, why not? If so, why?

Exercise 15.34:
Modify your debug function to let users turn debugging on or off. Implement the control two ways:

a. By defining a parameter to the debug function
b. By defining a class data member that allows individual objects to turn on or turn off the display of debugging information

Section 15.8.3: Using the Handle

Using Sales_item objects, we could more easily write our bookstore application. Our code wouldn’t need to manage pointers to the Item_base objects, yet the code would obtain virtual behavior on calls made through a Sales_item.

As an example, we could use Item_base objects to solve the problem proposed in Section 15.7 (p. 597). We could use Sales_items to keep track of the purchases a customer makes, storing a Sales_item representing each purchase in a multiset. When the customer was done shopping, we would total the sale.
Section : Comparing Two Sales_items

Before writing the function to total a sale, we need to define a way to compare Sales_items. To use Sales_item as the key in an associative container, we must be able to compare them (Section 10.3.1, p. 360). By default, the associative containers use the less-than operator on the key type. However, for the same reasons discussed about our original Sales_item type in Section 14.3.2 (p. 520), defining operator< for the Sales_item handle would be a bad idea: We want to take only the ISBN into account when we use Sales_item as a key, but want to consider all data members when determining equality.

Fortunately, the associative containers allow us to specify a function (or function object (Section 14.8, p. 530)) to use as the comparison function. We do so similarly to the way we passed a separate function to the stable_sort algorithm in Section 11.2.3 (p. 403). In that case, we needed only to pass an additional argument to stable_sort to provide a comparison function to use in place of the < operator. Overriding an associative container’s comparison function is a bit more complicated because, as we shall see, we must supply the comparison function when we define the container object.

Let’s start with the easy part, which is to define a function to use to compare Sales_item objects:

```cpp
// compare defines item ordering for the multiset in Basket
inline bool compare(const Sales_item &lhs, const Sales_item &rhs)
{
    return lhs->book() < rhs->book();
}
```

Our compare function has the same interface as the less-than operator. It returns a bool and takes two const references to Sales_items. It compares the parameters by comparing their ISBNs. This function uses the Sales_item -> operator, which returns a pointer to an Item_base object. That pointer is used to fetch and run the book member, which returns the ISBN.

Section : Using a Comparator with an Associative Container

If we think a bit about how the comparison function is used, we’ll realize that it must be stored as part of the container. The comparison function is used by any operation that adds or finds an element in the container. In principle, each of these operations could take an optional extra argument that represented the comparison function. However, this strategy would be error-prone: If two operations used different comparison functions, then the ordering would be inconsistent. It’s impossible to predict what would happen in practice.

To work effectively, an associative container needs to use the same comparison function for every operation. Yet, it is unreasonable to expect users to remember the comparison function every time, especially when there is no way to check that each call uses the same comparison function. Therefore, it makes sense for the container to remember the comparison function. By storing the comparator in the container object, we are assured that every operation that compares elements will do so consistently.
For the same reasons that the container needs to know the element type, it needs to know the comparator type in order to store the comparator. In principle, the container could infer this type by assuming that the comparator is pointer to a function that returns a `bool` and takes references to two objects of the `key_type` of the container. Unfortunately, this inferred type would be overly restrictive. For one thing, we should allow the comparator to be a function object as well as a plain function. Even if we were willing to require that the comparator be a function, the inferred type would still be too restrictive. After all, the comparison function might return an `int` or any other type that can be used in a condition. Similarly, the parameter type need not exactly match the `key_type`. Any parameter type that is convertible to the `key_type` should also be allowed.

So, to use our `Sales_item` comparison function, we must specify the comparator type when we define the `multiset`. In our case, that type is a function that returns a `bool` and takes two `const Sales_item` references.

We’ll start by defining a typedef that is a synonym for this type (Section 7.9, p. 276):

```cpp
// type of the comparison function used to order the multiset
typedef bool (*Comp)(const Sales_item&, const Sales_item&);
```

This statement defines `Comp` as a synonym for the pointer to function type that matches the comparison function we wish to use to compare `Sales_item` objects.

Next we’ll need to define a `multiset` that holds objects of type `Sales_item` and that uses this `Comp` type for its comparison function. Each constructor for the associative containers allows us to supply the name of the comparison function. We can define an empty `multiset` that uses our `compare` function as follows:

```cpp
std::multiset<Sales_item, Comp> items(compare);
```

This definition says that `items` is a `multiset` that holds `Sales_item` objects and uses an object of type `Comp` to compare them. The `multiset` is empty—we supplied no elements—but we did supply a comparison function named `compare`. When we add or look for elements in `items` our `compare` function will be used to order the `multiset`.

**Section : Containers and Handle Classes**

Now that we know how to supply a comparison function, we’ll define a class, named `Basket`, to keep track of a sale and calculate the purchase price:

```cpp
class Basket {
    // type of the comparison function used to order the multiset
    typedef bool (*Comp)(const Sales_item&, const Sales_item&);

    public:
    // make it easier to type the type of our set
    typedef std::multiset<Sales_item, Comp> set_type;
    // typedefs modeled after corresponding container types
    typedef set_type::size_type size_type;
    typedef set_type::const_iterator const_iter;

    Basket(): items(compare) { } // initialize the comparator
    void add_item(const Sales_item &item)
    { items.insert(item); }

    size_type size(const Sales_item &i) const
    { return items.count(i); }

    double total() const; // sum of net prices for all items in the basket

    private:
    std::multiset<Sales_item, Comp> items;
}
```
This class holds the customer’s purchases in a multiset of Sales_item objects. We use a multiset to allow the customer to buy multiple copies of the same book. The class defines a single constructor, the Basket default constructor. The class needs its own default constructor to pass compare to the multiset constructor that builds the items member.

The operations that the Basket class defines are fairly simple: add_item takes a reference to a Sales_item and puts a copy of that item into the multiset; item_count returns the number of records for this ISBN in the basket for a given ISBN. In addition to the operations, Basket defines three typedefs to make it easier to use its multiset member.

Section : Using the Handle to Execute a Virtual Function

The only complicated member of class Basket is the total function, which returns the price for all the items in the basket:

```cpp
double Basket::total() const
{
    double sum = 0.0; // holds the running total
    /* find each set of items with the same isbn and calculate
    * the net price for that quantity of items
    * iter refers to first copy of each book in the set
    * upper_bound refers to next element with a different isbn
    */
    for (const_iter iter = items.begin(); iter != items.end(); iter =
         items.upper_bound(*iter))
    {
        // we know there's at least one element with this key in
        // virtual call to net_price applies appropriate
discounts, if any
        sum += (*iter)->net_price(items.count(*iter));
    }
    return sum;
}
```

The total function has two interesting parts: the call to the net_price function, and the structure of the for loop. We’ll look at each in turn.

When we call net_price, we need to tell it how many copies of a given book are being purchased. The net_price function uses this argument to determine whether the purchase qualifies for a discount. This requirement implies that we’d like to process the multiset in chunks—processing all the records for a given title in one chunk and then the set of those for the next title and so on. Fortunately, multiset is well suited to this problem.

Our for loop starts by defining and initializing iter to refer to the first element in the multiset. We use the multiset count member (Section 10.3.6, p. 367) to determine how many elements in the multiset have the same key (e.g., same isbn) and use that number as the argument to the call to net_price.
The interesting bit is the “increment” expression in the for. Rather than the usual loop that reads each element, we advance iter to refer to the next key. We skip over all the elements that match the current key by calling upper_bound (Section 10.5.2, p. 377). The call to upper_bound returns the iterator that refers to the element just past the last one with the same key as in iter. That iterator we get back denotes either the end of the set or the next unique book. We test the new value of iter. If iter is equal to items.end(), we drop out of the for. Otherwise, we process the next book.

The body of the for calls the net_price function. That call can be a bit tricky to read:

```cpp
    sum += (*iter)->net_price(items.count(*iter));
```

We dereference iter to get the underlying Sales_item to which we apply the overloaded arrow operator from the Sales_item class. That operator returns the underlying Item_base object to which the handle is attached. From that object we call net_price, passing the count of items with the same isbn. The net_price function is virtual, so the version of the pricing function that is called depends on the type of the underlying Item_base object.

Exercise 15.35:
Write your own version of the compare function and Basket class and use them to manage a sale.

Exercise 15.36:
What is the underlying type of Basket::const_iter?

Exercise 15.37:
Why did we define the Comp typedef in the private part of Basket?

Exercise 15.38:
Why did we define two private sections in Basket?

Section 15.9: Text Queries Revisited

As a final example of inheritance, we’ll extend our text query application from Section 10.6 (p. 379). The class we developed there let us look for occurrences of a given word in a text file. We’d like to extend the system to support more complex queries.

For illustration purposes, we’ll run queries against the following simple story:

Alice Emma has long flowing red hair.
Her Daddy says when the wind blows
through her hair, it looks almost alive,
like a fiery bird in flight.
A beautiful fiery bird, he tells her,
magical but untamed.
"Daddy, shush, there is no such thing,"
she tells him, at the same time wanting
him to tell her more.
Shyly, she asks, "I mean, Daddy, is there?"

Our system should support:

1. Word queries that find a single word. All lines in which the word appears should be displayed in ascending order:
2. Executed Query for:
3. Daddy match occurs 3 times:
4. (line 2) Her Daddy says when the wind blows
5. (line 7) "Daddy, shush, there is no such thing,"
6. (line 10) Shyly, she asks, "I mean, Daddy, is there?"
7. Not queries, using the ~ operator. All lines that do not match the query are displayed:
8. Executed Query for: ~(Alice)
9. match occurs 9 times:
10. (line 2) Her Daddy says when the wind blows
11. (line 3) through her hair, it looks almost alive,
12. (line 4) like a fiery bird in flight. ...
13. Or queries, using the | operator. All lines in which either of two queries match are displayed:
14. Executing Query for: (hair | Alice)
15. match occurs 2 times:
16. (line 1) Alice Emma has long flowing red hair.
17. (line 3) through her hair, it looks almost alive,
18. And queries, using the & operator. All lines in which both queries match are displayed.
19. Executed query: (hair & Alice)
20. match occurs 1 time:
21. (line 1) Alice Emma has long flowing red hair.

Moreover, these elements can be combined, as in
fiery & bird | wind

Our system will not be sophisticated enough to read these expressions. Instead, we’ll build them up inside a C++ program. Hence, we’ll evaluate compound expressions such as this example using normal C++ precedence rules. The evaluation of this query will match a line in which fiery and bird appear or one in which wind appears. It will not match a line on which fiery or bird appears alone:
Executing Query for: (((fiery & bird) | wind)
macht occurs 3 times:
(line 2) Her Daddy says when the wind blows
(line 4) like a fiery bird in flight.
(line 5) A beautiful fiery bird, he tells her,

Our output will print the query, using parentheses to indicate the way in which the query was interpreted. As with our original implementation, our system must be smart enough not to display the same line more than once.

Section 15.9.1: An Object-Oriented Solution

We might think that we could use the TextQuery class from page 382 to represent our word queries. We might then derive our other queries from that class. However, this design would be flawed. Conceptually, a “not” query is not a kind of word query. Instead, a not query “has a” query (word query or any other kind of query) whose value it negates.

This observation suggests that we model our different kinds of queries as independent classes that share a common base class:

WordQuery // Shakespeare
NotQuery // ~Shakespeare
OrQuery // Shakespeare | Marlowe
AndQuery // William & Shakespeare
Instead of inheriting from `TextQuery`, we will use that class to hold the file and build the associated `word_map`. We’ll use the query classes to build up expressions that will ultimately run queries against the file in a `TextQuery` object.

Section : Abstract Interface Class

We have identified four kinds of query classes. These classes are conceptually siblings. Each class shares the same abstract interface, which suggests that we’ll need to define an abstract base class (Section 15.6, p. 595) to represent the operations performed by a query. We’ll name our abstract class `Query_base`, indicating that its role is to serve as the root of our query hierarchy.

We’ll derive `WordQuery` and `NotQuery` directly from our abstract base. The `AndQuery` and `OrQuery` classes share one property that the other classes in our system do not: They each have two operands. To model this fact, we’ll add another abstract class, named `BinaryQuery`, to our hierarchy to represent queries with two operands. The `AndQuery` and `OrQuery` classes will inherit from the `BinaryQuery` class, which in turn will inherit from `Query_base`. These decisions give us the class design represented in Figure 15.3 on the next page.

![Query_base Inheritance Hierarchy](#)

Section : Operations

Our `Query_base` classes exist mostly to represent kinds of queries; they do little actual work. We’ll reuse our `TextQuery` class to store the file, build the query map, and search for each word. Our query types need only two operations:

1. An `eval` operation to return the set of matching line numbers. This operation takes a `TextQuery` object on which to execute the query.
2. A `display` operation that takes a reference to an `ostream` and prints the query that a given object performs on that stream.

We’ll define each of these operations as pure virtual functions (Section 15.6, p. 595) in the `Query_base` class. Each of our derived classes will have to define its own version of these functions.
Section 15.9.2: A Valuelike Handle

Our program will deal with evaluating queries, not with building them. However, we need to be able to create queries in order to run our program. The simplest way to do so is to write C++ expressions to create queries directly. For example, we’d like to be able to write code such as

```cpp
Query q = Query("fiery") & Query("bird") | Query("wind");
```

to generate the compound query previously described.

This problem description implicitly suggests that user-level code won’t use our inherited classes directly. Instead, we’ll define a handle class named `Query`, which will hide the hierarchy. User code will execute in terms of the handle; user code will only indirectly manipulate `Query_base` objects.

As with our `Sales_item` handle, our `Query` handle will hold a pointer to an object of a type in an inheritance hierarchy. The `Query` class will also point to a use count, which we’ll use to manage the object to which the handle points.

In this case, our handle will completely hide the underlying inheritance hierarchy. Users will create and manipulate `Query_base` objects only indirectly through operations on `Query` objects. We’ll define three overloaded operators on `Query` objects and a `Query` constructor that will dynamically allocate a new `Query_base` object. Each operator will bind the generated `Query_base` object to a `Query` handle: The `&` operator will generate a `Query` bound to a new `AndQuery`; the `|` operator will generate a `Query` bound to a new `OrQuery`; and the `~` operator will generate a `Query` bound to a new `NotQuery`. We’ll give `Query` a constructor that takes a `string`. This constructor will generate a new `WordQuery`.

The `Query` class will provide the same operations as the `Query_base` classes: `eval` to evaluate the associated query, and `display` to print the query. It will define an overloaded output operator to display the associated query.

<table>
<thead>
<tr>
<th>Table 15.1: Query Program Design: A Recap</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TextQuery</strong></td>
</tr>
<tr>
<td><strong>Query_base</strong></td>
</tr>
<tr>
<td><strong>Query</strong></td>
</tr>
<tr>
<td><strong>WordQuery</strong></td>
</tr>
<tr>
<td><strong>NotQuery</strong></td>
</tr>
<tr>
<td><strong>BinaryQuery</strong></td>
</tr>
<tr>
<td><strong>OrQuery</strong></td>
</tr>
<tr>
<td><strong>AndQuery</strong></td>
</tr>
</tbody>
</table>
q1 & q2 \quad \text{Returns a Query bound to a new AndQuery object that holds } q1 \text{ and } q2.

q1 | q2 \quad \text{Returns a Query bound to a new OrQuery object that holds } q1 \text{ and } q2.

~q \quad \text{Returns a Query bound to a new NotQuery object that holds } q.

Query \ q(s) \quad \text{Binds the Query } q \text{ to a new WordQuery that holds the string } s.

Section : Our Design: A Recap

It is often the case, especially when new to designing object-oriented systems, that understanding the design is the hardest part. Once we’re comfortable with the design, the implementation flows naturally.

It is important to realize that much of the work in this application consists of building objects to represent the user’s query. As illustrated in Figure 15.4 on the following page, an expression such as

\text{Query } q = \text{Query("fiery") } & \text{ Query("bird") } | \text{ Query("wind")};

generates ten objects: five Query_base objects and their associated handles. The five \text{Query_base} objects are three WordQueries, an OrQuery, and an AndQuery.

Each small square represents a Query object

Objects Created by the Expression
\text{Query("fiery") } & \text{ Query("bird") } | \text{ Query("wind")};

Figure 15.4: Objects Created by Query Expressions

Once the tree of objects is built up, evaluating (or displaying) a given query is basically a process (managed for us by the compiler) of following these links, asking each object in the tree to evaluate (or display) itself. For example, if we call eval on q (i.e., on the root of this tree), then eval will ask the OrQuery to which it points to eval itself. Evaluating this OrQuery calls eval on its two operands, which in turn calls eval on the AndQuery and WordQuery that looks for the word wind, and so on.

Exercise 15.39:
Given that \text{s1}, \text{s2}, \text{s3} and \text{s4} are all strings, determine what objects are created in the following uses of the Query class:
Section 15.9.3: The \texttt{Query}\_base Class

Now that we’ve explained our design, we’ll start our implementation by defining the \texttt{Query}\_base class:

```cpp
// private, abstract class acts as a base class for concrete query types
class Query_base {
    friend class Query;
protected:
    typedef TextQuery::line_no line_no;
    virtual ~Query_base() = 0;
private:
    // eval returns the |set| of lines that this Query matches
    virtual std::set<line_no> eval(const TextQuery&) const = 0;
    // display prints the query
    virtual std::ostream& display(std::ostream& = std::cout) const = 0;
};
```

The class defines two interface members: \texttt{eval} and \texttt{display}. Both are pure virtual functions (Section 15.6, p. 595), which makes this class abstract. There will be no objects of type \texttt{Query}\_base in our applications.

Users and the derived classes will use the \texttt{Query}\_base class only through the \texttt{Query} handle. Therefore, we made our \texttt{Query}\_base interface private. The (virtual) destructor (Section 15.4.4, p. 587) and the typedef are protected so that the derived types can access these members. The destructor is used (implicitly) by the derived-class destructors and so must be accessible to them.

We grant friendship to the \texttt{Query} handle class. Members of that class will call the virtuals in \texttt{Query}\_base and so must have access to them.

Section 15.9.4: The \texttt{Query} Handle Class

Our \texttt{Query} handle will be similar to the \texttt{Sales}\_item class in that it will hold a pointer to the \texttt{Query}\_base and a pointer to a use count. As in the \texttt{Sales}\_item class, the copy-control members of \texttt{Query} will manage the use count and the \texttt{Query}\_base pointer. Unlike the \texttt{Sales}\_item class, \texttt{Query} will provide the only interface to the \texttt{Query}\_base hierarchy. Users will not directly access any of the members of \texttt{Query}\_base or its derived classes. This design decision leads to two differences between \texttt{Query} and \texttt{Sales}\_item. The first is that the \texttt{Query} class won’t define overloaded versions of dereference and arrow operators. The \texttt{Query}\_base class has no public members. If the \texttt{Query} handle defined the dereference or arrow operators, they would be of no use! Any attempt to use those operators to access a \texttt{Query}\_base member would fail. Instead, \texttt{Query} must define its own versions of the \texttt{Query}\_base interface functions \texttt{eval} and \texttt{display}. 

(a) Query(s1) \texttt{\mid} Query(s2) \texttt{\&\&} \lnot \texttt{Query(s3)};
(b) Query(s1) \texttt{\mid} (Query(s2) \texttt{\&\&} \lnot \texttt{Query(s3)});
(c) (Query(s1) \texttt{\&}\texttt{\&} (Query(s2)) \texttt{\mid} (Query(s3) \texttt{\&\&} \texttt{Query(s4)}));
The other difference results from how we intend objects of the hierarchy to be created. Our design says that objects derived from *Query_base* will be created only through operations on the *Query* handle. This difference results in different constructors being required for the *Query* class than were used in the *Sales_item* handle.

**Section : The *Query* Class**

Given the preceding design, the *Query* class itself is quite simple:

```cpp
// handle class to manage the Query_base inheritance hierarchy
class Query {
  // these operators need access to the Query_base* constructor
  friend Query operator ~(const Query &);
  friend Query operator |(const Query&, const Query&);
  friend Query operator & (const Query&, const Query&);

  public:
    Query(const std::string&); // builds a new WordQuery
    Query(const Query &q): q(q.q), use(q.use) { ++*use; }
  ~Query() { decr_use(); }
  Query& operator=(const Query&);

  // interface functions: will call corresponding Query_base operations
  std::set<TextQuery::line_no> eval(const TextQuery &t) const {
    return q->eval(t); }
  std::ostream &display(std::ostream &os) const {
    return q->display(os); }

  private:
    Query(Query_base *query): q(query),
      use(new std::size_t(1)) { }
    Query_base *q;
    std::size_t *use;
    void decr_use() {
      if (--*use == 0) { delete q; delete use; } } }
};
```

We start by naming as friends the operators that create *Query* objects. We’ll see shortly why these operators need to be friends.

In the public interface for *Query*, we declare, but cannot yet define, the constructor that takes a string. That constructor creates a *WordQuery* object, so we cannot define the constructor until we have defined the *WordQuery* class.

The next three members handle copy control and are the same as the corresponding members of the *Sales_item* class.

The last two public members represent the interface for *Query_base*. In each case, the *Query* operation uses its *Query_base* pointer to call the respective *Query_base* operation. These operations are virtual. The actual version that is called is determined at run time and will depend on the type of the object to which *q* points.

The private implementation of *Query* includes a constructor that takes a pointer to a *Query_base* object. This constructor stores in *q* the pointer it is given and allocates a new use counter, which it initializes to one. This constructor is private because we don’t intend general user code to define *Query_base* objects. Instead, the constructor is needed
for the operators that create Query objects. Because the constructor is private, the operators had to be made friends.

**Section: The Query Overloaded Operators**

The |, & and ~ operators create OrQuery, AndQuery, and NotQuery objects, respectively:

```cpp
inline Query operator&(const Query &lhs, const Query &rhs)
{
    return new AndQuery(lhs, rhs);
}
inline Query operator|(const Query &lhs, const Query &rhs)
{
    return new OrQuery(lhs, rhs);
}
inline Query operator~(const Query &oper)
{
    return new NotQuery(oper);
}
```

Each of these operations dynamically allocates a new object of a type derived from Query_base. The return (implicitly) uses the Query constructor that takes a pointer to a Query_base to create the Query object from the Query_base pointer that the operation allocates. For example the return statement in the ~ operator is equivalent to

```cpp
// allocate a new Not Query object
// convert the resulting pointer to NotQuery to a pointer to Query_base
Query_base *tmp = new NotQuery(expr);
return Query(tmp); // use Query constructor that takes a pointer to Query_base
```

There is no operator to create a WordQuery. Instead, we gave our Query class a constructor that takes a string. That constructor generates a WordQuery to look for the given string.

**Section: The Query Output Operator**

We’d like users to be able to print Querys using the normal (overloaded) output operator. However, we also need the print operation to be virtual—printing a Query should print the Query_base object to which the Query points. There’s only one problem: only member functions can be virtual, but the output operator cannot be a member of the Query_base classes (Section 14.2.1, p. 514).

To obtain the necessary virtual behavior, our Query_base classes defined a virtual display member, which the Query output operator will use:

```cpp
inline std::ostream&
operator<<(std::ostream &os, const Query &q)
{
    return q.display(os);
}
```

When we write

```cpp
Query andq = Query(sought1) & Query(sought2);
```
cout << "\nExecuted query: " << andq << endl;
the Query output operator is invoked. That operator calls
q.display(os)
with q referring to the Query object that points to this AndQuery, an dos bound to cout.
When we write
Query name(sought);
cout << "\nExecuted Query for: " << name << endl;
the WordQuery instance of display is called. More generally, a call
Query query = some_query;
cout << query << endl;
invokes the instance of display associated with the object that query addresses at that
point in the execution of our program.

Section 15.9.5: The Derived Classes

We next need to implement our concrete query classes. The one interesting part about
these classes is how they are represented. The WordQuery class is most straightforward.
Its job is to hold the search word.
The other classes operate on one or two Query operands. A NotQuery negates the result
of another Query. Both AndQuery and OrQuery have two operands, which are actually
stored in their common base class, BinaryQuery.
In each of these classes, the operand(s) could be an object of any of the concrete
Query_base classes: A NotQuery could be applied to a WordQuery, an AndQuery, an
OrQuery, or another NotQuery. To allow this flexibility, the operands must be stored as
pointers to Query_base that might point to any one of the concrete Query_base classes.
However, rather than storing a Query_base pointer, our classes will themselves use the
Query handle. Just as user code is simplified by using a handle, we can simplify our own
class code by using the same handle class. We’ll make the Query operand const because
once a given Query_base object is built, there are no operations that can change the
operand(s).
Now that we know the design for these classes, we can implement them.

Section : The WordQuery Class

A WordQuery is a kind of Query_base that looks for a specified word in a given query
map:
class WordQuery: public Query_base {
  friend class Query; // Query uses the WordQuery constructor
  WordQuery(const std::string &s): query_word(s) { }
  // concrete class: WordQuery defines all inherited pure
  virtual functions
  std::set<line_no> eval(const TextQuery &t) const
    { return t.run_query(query_word); }
  std::ostream& display (std::ostream &os) const
    { return os << query_word; }
  std::string query_word; // word for which to search
};
Like `Query_base`, `WordQuery` has no public members; `WordQuery` must make `Query` a friend to allow `Query` to access the `WordQuery` constructor.

Each of the concrete query classes must define the inherited pure virtual functions. The `WordQuery` operations are simple enough to define in the class body. The `eval` member calls the `query_text` member of its `TextQuery` parameter passing it the string that was used to create this `WordQuery`. To display a `WordQuery`, we print the `query_word`.

**Section: The `NotQuery` Class**

A `NotQuery` holds a `const Query`, which it negates:
```cpp
class NotQuery: public Query_base {
    friend Query operator~(const Query &);
    NotQuery(Query q): query(q) { }
    // concrete class: NotQuery defines all inherited pure
    virtual functions
    std::set<line_no> eval(const TextQuery&) const;
    std::ostream& display(std::ostream &os) const
    { return os << "~(" << query << ")"; }
    const Query query;
};
```
The `Query` overloaded `~` operator is made a friend to allow that operator to create a new `NotQuery` object. To display a `NotQuery`, we print the `~` symbol followed by the underlying `Query`. We parenthesize the output to ensure that precedence is clear to the reader.

The use of the output operator in the `display` operation is ultimately a virtual call to a `Query_base` object:
```cpp
// uses the Query output operator, which calls Query::display
// that function makes a virtual call to Query_base::display
{ return os << "~(" << query << ")" }
```
The `eval` member is complicated enough that we will implement it outside the class body. The `eval` function appears in [Section 15.9.6 (p. 620)](#).

**Section: The `BinaryQuery` Class**

The `BinaryQuery` class is an abstract class that holds the data needed by the two query types, `AndQuery` and `OrQuery`, that operate on two operands:
```cpp
class BinaryQuery: public Query_base {
    protected:
    BinaryQuery(Query left, Query right, std::string op):
        lhs(left), rhs(right), oper(op) { }
    // abstract class: BinaryQuery doesn't define eval
    std::ostream& display(std::ostream &os) const
    { return os << "(" << lhs << " " << oper << " "
        << rhs << ")"; }
    const Query lhs, rhs; // right- and left-hand operands
    const std::string oper; // name of the operator
};
```
The data in a BinaryQuery are the two Query operands and the operator symbol to use when displaying the query. These data are all declared const, because the contents of a query should not change once it has been constructed. The constructor takes the two operands and the operator symbol, which it stores in the appropriate data members.

To display a BinaryOperator, we print the parenthesized expression consisting of the left-hand operand, followed by the operator, followed by the right-hand operand. As when we displayed a NotQuery, the overloaded << operator that is used to print left and right ultimately makes a virtual call to the underlying Query_base display.

The BinaryQuery class does not define the eval function and so inherits a pure virtual. As such, BinaryQuery is also an abstract class, and we cannot create objects of BinaryQuery type.

Section : The AndQuery and OrQuery Classes

The AndQuery and OrQuery classes are nearly identical:

```cpp
class AndQuery: public BinaryQuery {
    friend Query operator&(const Query&, const Query&);
    AndQuery (Query left, Query right)
    : BinaryQuery(left, right, "&") { }
    // concrete class: AndQuery inherits display and defines remaining pure virtual
    std::set<line_no> eval(const TextQuery&) const;
};

class OrQuery: public BinaryQuery {
    friend Query operator|(const Query&, const Query&);
    OrQuery (Query left, Query right)
    : BinaryQuery(left, right, "|") { }
    // concrete class: OrQuery inherits display and defines remaining pure virtual
    std::set<line_no> eval(const TextQuery&) const;
};
```

These classes make the respective operator a friend and define a constructor to create their BinaryQuery base part with the appropriate operator. They inherit the BinaryQuery definition of display, but each defines its own version of the eval function.

Exercise 15.40:
For the expression built in Figure 15.4 (p. 612)

a. List the constructors executed in processing this expression.
b. List the calls to display and to the overloaded << operator that are made in executing cout << q.
c. List the calls to eval made when evaluating q.eval.

Section 15.9.6: The eval Functions

The heart of the query class hierarchy are the eval virtual functions. Each of these functions calls eval on its operand(s) and then applies its own logic: The AndQuery
eval operation returns the union of the results of its two operands; OrQuery returns the intersection. The NotQuery is more complicated: It must return the line numbers not in its operand’s set.

Section: OrQuery::eval

An OrQuery merges the set of line numbers returned by its two operands—its result is the union of the results for its two operands:

```cpp
// returns union of its operands' result sets
set<TextQuery::line_no> OrQuery::eval(const TextQuery& file) const
{
    // virtual calls through the Query handle to get result sets for the operands
    set<line_no> right = rhs.eval(file),
                 ret_lines = lhs.eval(file); // destination to hold results
    // inserts the lines from right that aren't already in ret_lines
    ret_lines.insert(right.begin(), right.end());
    return ret_lines;
}
```

The eval function starts by calling eval on each of its Query operands. Those calls call Query::eval, which in turn makes a virtual call to eval on the underlying Query_base object. Each of these calls yields a set of line numbers in which its operand appears. We then call `insert` on `ret_lines`, passing a pair of iterators denoting the set returned from evaluating the right-hand operand. Because `ret_lines` is a set, this call adds the elements from right that are not also in left into `ret_lines`. After the call to `insert`, `ret_lines` contains each line number that was in either of the left or right sets. We complete the function by returning `ret_lines`.

Section: AndQuery::eval

The AndQuery version of eval uses one of the library algorithms that performs setlike operations. These algorithms are described in the Library Appendix, in Section A.2.8 (p. 821):

```cpp
// returns intersection of its operands' result sets
set<TextQuery::line_no> AndQuery::eval(const TextQuery& file) const
{
    // virtual calls through the Query handle to get result sets for the operands
    set<line_no> left = lhs.eval(file),
                 right = rhs.eval(file);
    set<line_no> ret_lines; // destination to hold results
    // writes intersection of two ranges to a destination iterator
    // destination iterator in this call adds elements to ret
    set_intersection(left.begin(), left.end(),
                     right.begin(), right.end(),
                     ret_lines.begin(), ret_lines.end());
    return ret_lines;
}
```
inserter(ret_lines, ret_lines.begin()));
    return ret_lines;
}

This version of eval uses the set_intersection algorithm to find the lines in common to both queries: That algorithm takes five iterators: The first four denote two input ranges, and the last denotes a destination. The algorithm writes each element that is in both of the two input ranges into the destination. The destination in this call is an insert iterator (Section 11.3.1, p. 406) which inserts new elements into ret_lines.

Section : NotQuery::eval

NotQuery finds each line of the text within which the operand is not found. To support this function, we need the TextQuery class to add a member to return the size of the file, so that we can know what line numbers exist.

    // returns lines not in its operand's result set
    set<TextQuery::line_no> NotQuery::eval(const TextQuery& file) const
    {
        // virtual call through the Query handle to eval
        set<TextQuery::line_no> has_val = query.eval(file);
        set<line_no> ret_lines;
        // for each line in the input file, check whether that line
        // is in has_val
        // if not, add that line number to ret_lines
        for (TextQuery::line_no n = 0; n != file.size(); ++n)
            if (has_val.find(n) == has_val.end())
                ret_lines.insert(n);
        return ret_lines;
    }

As in the other eval functions, we start by calling eval on this object’s operand. That call returns the set of line numbers on which the operand appears. What we want is the set of line numbers on which the operand does not appear. We obtain that set by looking at each line number in the input file. We use the size member that must be added to TextQuery to control the for loop. That loop adds each line number to ret_lines that does not appear in has_val. Once we’ve processed all the line numbers, we return ret_lines.

Exercise 15.41:
Implement the Query and Query_base classes, and add the needed size operation to the TextQuery class from Chapter 10. Test your application by evaluating and printing a query such as the one in Figure 15.4 (p. 612).

Exercise 15.42:
Design and implement one of the following enhancements:

a. Introduce support for evaluating words based on their presence within the same sentence rather than the same line.

b. Introduce a history system in which the user can refer to a previous query by number, possibly adding to it or combining it with another.
c. Rather than displaying the count of matches and all the matching lines, allow the user to indicate a range of lines to display, both for intermediate query evaluation and the final query.

Section : Chapter Summary

The ideas of inheritance and dynamic binding are simple but powerful. Inheritance lets us write new classes that share behavior with their base class(es) but redefine that behavior as needed. Dynamic binding lets the compiler decide at run time which version of a function to run based on an object’s dynamic type. The combination of inheritance and dynamic binding lets us write type-independent programs that have type-specific behavior.

In C++, dynamic binding applies only to functions declared as virtual when called through a reference or pointer. It is common for C++ programs to define handle classes to interface to an inheritance hierarchy. These classes allocate and manage pointers to objects in the inheritance hierarchy, thus obtaining dynamic behavior while shielding user code from having to deal with pointers.

Inherited objects are composed of base-class part(s) and a derived-class part. Inherited objects are constructed, copied, and assigned by constructing, copying, and assigning the base part(s) of the object before handling the derived part. Because a derived object contains a base part, it is possible to convert a reference or pointer to a derived type to a reference or pointer to its base type.

Base classes usually should define a virtual destructor even if the class otherwise has no need for a destructor. The destructor must be virtual if a pointer to a base is ever deleted when it actually addresses a derived-type object.

Section : Defined Terms

abstract base class

Class that has or inherits one or more pure virtual functions. It is not possible to create objects of an abstract base-class type. Abstract base classes exist to define an interface. Derived classes will complete the type by defining type-specific implementations for the pure virtuals defined in the base.

base class

Class from which another class inherits. The members of the base class become members of the derived class.

class derivation list

Used by a class definition to indicate that the class is a derived class. A derivation list includes an optional access level and names the base class. If no access label
is specified, the type of inheritance depends on the keyword used to define the derived class. By default, if the derived class is defined with the `struct` keyword, then the base class is inherited `publicly`. If the class is defined using the `class` keyword, then the base class is inherited `privately`.

**derived class**

A class that inherits from another class. The members of the base class are also members of the derived class. A derived class can redefine the members of its base and can define new members. A derived-class scope is nested in the scope of its base class(es), so the derived class can access members of the base class directly. Members defined in the derived with the same name as members in the base hide those base members; in particular, member functions in the derived do not overload members from the base. A hidden member in the base can be accessed using the scope operator.

**direct base class**

Synonym for immediate base class.

**dynamic binding**

Delaying until run time the selection of which function to run. In C++, dynamic binding refers to the run-time choice of which `virtual` function to run based on the underlying type of the object to which a reference or pointer is bound.

**dynamic type**

Type at run time. Pointers and references to base-class types can be bound to objects of derived type. In such cases the static type is reference (or pointer) to base, but the dynamic type is reference (or pointer) to derived.

**handle class**

Class that provides an interface to another class. Commonly used to allocate and manage a pointer to an object of an inheritance hierarchy.

**immediate base class**

A base class from which a derived class inherits directly. The immediate base is the class named in the derivation list. The immediate base may itself be a derived class.

**indirect base class**
A base class that is not immediate. A class from which the immediate base class inherits, directly or indirectly, is an indirect base class to the derived class.

**inheritance hierarchy**

Term used to describe the relationships among classes related by inheritance that share a common base class.

**object-oriented programming**

Term used to describe programs that use data abstraction, inheritance, and dynamic binding.

**polymorphism**

A term derived from a Greek word that means “many forms.” In object-oriented programming, polymorphism refers to the ability to obtain type-specific behavior based on the dynamic type of a reference or pointer.

**private inheritance**

A form of implementation inheritance in which the public and protected members of a private base class are private in the derived.

**protected access label**

Members defined after a protected label may be accessed by class members and friends and by the members (but not friends) of a derived class. protected members are not accessible to ordinary users of the class.

**protected inheritance**

In protected inheritance the protected and public members of the base class are protected in the derived class.

**public inheritance**

The public interface of the base class is part of the public interface of the derived class.

**pure virtual**

A virtual function declared in the class header using =0 at the end of the function’s parameter list. A pure virtual is one that need not be (but may be) defined by the class. A class with a pure virtual is an abstract class. If a derived
class does not define its own version of an inherited pure virtual, then the derived class is abstract as well.

refactoring

Redesigning programs to collect related parts into a single abstraction, replacing the original code by uses of the new abstraction. In OO programs, refactoring frequently happens when redesigning the classes in an inheritance hierarchy. Refactoring often occurs in response to a change in requirements. In general, classes are refactored to move data or function members to the highest common point in the hierarchy to avoid code duplication.

sliced

Term used to describe what happens when an object of derived type is used to initialize or assign an object of the base type. The derived portion of the object is “sliced down,” leaving only the base portion, which is assigned to the base.

static type

Compile-time type. Static type of an object is the same as its dynamic type. The dynamic type of an object to which a reference or pointer refers may differ from the static type of the reference or pointer.

virtual function

A member function that defines type-specific behavior. Calls to a virtual made through a reference or pointer are resolved at run time, based on the type of the object to which the reference or pointer is bound.