CHAPTER

4 Inheritance

As mentioned in Chapter 2, an important goal of object-oriented programming is code reuse. Just as engineers use components over and over in their designs, programmers should be able to reuse objects rather than repeatedly reimplementing them. In Chapter 3 we saw one mechanism for reuse provided by C++, the template. Templates are appropriate if the basic functionality of the code is type independent. The other mechanism for code reuse is *inheritance*. Inheritance allows us to extend the functionality of an object; in other words, we can create new types with restricted (or extended) properties of the original type. Inheritance goes a long way toward our goal for code reuse.

In this chapter, we will see:

- General principles of inheritance and the related object-oriented concept of polymorphism
- How inheritance is implemented in C++
- How a collection of classes can be derived from a single abstract class
- How run-time binding decisions, rather than compile time linking decisions, can be made for these classes

4.1 What is Inheritance?

Inheritance is the fundamental object-oriented principle that is used to reuse code among related classes. Inheritance models the *IS-A relationship*. In an IS-A relationship, we say the derived class *is a* (variation of the) base class. For example, a Circle IS-A Shape and a Car IS-A Vehicle. However, an Ellipse IS-NOT-A Circle. Inheritance relationships form *hierarchies*. For instance, we can extend Car to other classes, since a ForeignCar IS-A Car (and pays tariffs) and a DomesticCar IS-A Car (and does not pay tariffs), and so on.

In an *IS-A relationship*, we say the derived class *is a* (variation of the) base class.

Another type of relationship is a *HAS-A* (or IS-COMPOSED-OF) *relationship*. This type of relationship does not possess the properties that would be natural in an inheritance hierarchy. An example of a HAS-A relationship is that a car HAS-A steering wheel. Generally, HAS-A relationships should not be modeled by inheritance. Instead, they should use the technique of *composition*, in which the components are simply made private data fields.

In a *HAS-A relationship*, we say the derived class *has a* (instance of the) base class. *Composition* is used to model HAS-A relationships.

The C++ language itself makes some use of inheritance in implementing its class libraries. Two examples are exceptions and files:

- *Exceptions*. C++ defines, in `<stdexcept>`, the class `exception`.There are several kinds of exceptions, including `bad_alloc` and `bad_cast`. Figure 4.1 illustrates some of the classes in the `exception` hierarchy. We will explain the diagram shortly. Each of the classes is a separate class, but for all of them, the `what` method can be used to return a (primitive) string that details an error message.
- *I/O*. As we see in Figure 4.2, the streams hierarchy (`istream`, `ifstream`, etc.) uses inheritance. The streams hierarchy is also more complex that what is shown.

In addition, systems such as Visual C++ and Borland CBuilder provide class libraries that can be used to design graphical user interfaces (GUIs). These, libraries, which define components such as buttons, choice-lists, text-areas, and windows, (all in different flavors), make heavy use of inheritance.

In all cases, the inheritance models an IS-A relationship. A button IS-A component. A `bad_cast` IS-A `exception`. An `ifstream` IS-A `istream` (but not vice-versa!). Because of the IS-A relationship, the fundamental property of inheritance guarantees that any method that can be performed by `istream` can also be performed by `ifstream`, and an `ifstream` object can always be referenced by an `istream` reference. Note that the reverse is not true. This is why I/O operations are always written in terms of `istream` and `ostream`.

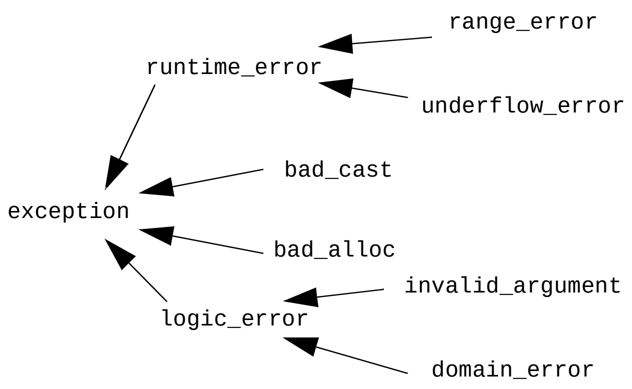

Figure 4.1 Part of the `exception` hierarchy

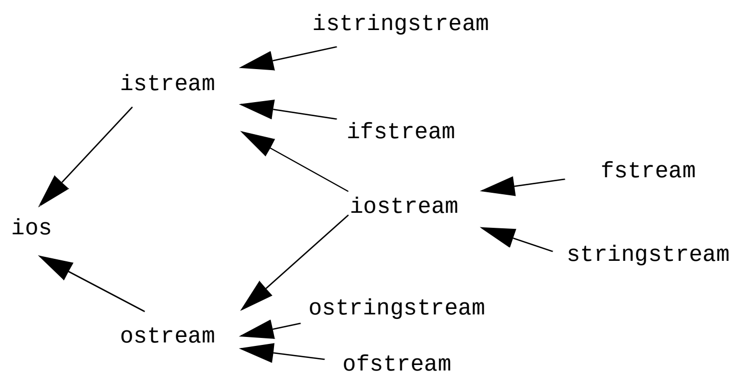

Figure 4.2 Part of the streams hierarchy

As a second example, since `what` is a method available in the `exception` class, if we need to catch exceptions defined in Figure 4.1 using a `catch` handler, we can always write[1]:

```
catch( const exception & e ) { cout << e.what( ) << endl; }
```

If `e` references a `bad_cast` object, the call to `e.what()` makes sense. This is because an `exception` object supports the `what` method, and a `bad_cast` IS-A `exception`, meaning that it supports at least as much as `exception`. Depending on the circumstances of the class hierarchy, the `what` method could be invariant or it could be *specialized* for each different class. When a method is invariant over a hierarchy, meaning it always has the same functionality for all classes in the hierarchy, we avoid having to rewrite an implementation of a class method.

The call to `what` also illustrates an important object-oriented principle known as *polymorphism*. A reference variable that is polymorphic can reference objects of several different types. When operations are applied to the reference, the operation that is appropriate to the actual referenced object is automatically selected. The same is true for pointer variables (remember that a reference really is a pointer). In the case of an `exception` reference, a run-time decision is made: the `what` method for the object that `e` actually references at run-time is the one that is used. This is known as *dynamic binding* or *late binding*. Unfortunately, although dynamic binding is the preferred behavior, it is not the default in C++. This language flaw leads to complications.

A *polymorphic* variable can reference objects of several different types. When operations are applied to the polymorphic variable, the operation appropriate to the referenced object is automatically selected.

1. Exceptions are handled by try/catch blocks. An illustration of the syntax is in Figure 4.7 on page 177. Code that might throw the exception is placed in a try block. The exception is handled in a catch block. Since the exception object is passed into the catch block, any public methods defined for the exception object can be used on it and any public data defined in the exception object can be examined.

Inheritance allows us to derive classes from a *base class* without disturbing the implementation of the base class.

In *inheritance*, we have a *base class* from which other classes are derived. The base class is the class on which the inheritance is based. A *derived class* inherits all the properties of a base class, meaning that all public methods available to the base class become public methods, with identical implementations for the derived class. It can then add data members and additional methods and change the meaning of the inherited methods. Each derived class is a completely new class. However, the base class is completely unaffected by any changes that are made in the derived class. Thus, in designing the derived class, it is impossible to break the base class. This greatly simplifies the task of software maintenance.

Each *derived class* is a completely new class that nonetheless has some compatibility with the class from which it was derived.

A derived class is type compatible with its base class, meaning that a reference variable of the base class type may reference an object of the derived class, but not vice versa (and similarly for pointers). Sibling classes (that is, classes derived from a common class) are not type compatible.

As mentioned earlier, the use of inheritance typically generates a hierarchy of classes. Figure 4.1 illustrated a small part of the `exception` hierarchy. Notice that `range_error` is indirectly, rather than directly, derived from `exception`. This fact is transparent to the user of the classes because IS-A relationships are transitive. In other words, if *X* IS-A *Y* and *Y* IS-A *Z*, then *X* IS-A *Z*. The `exception` hierarchy illustrates the typical design issues of factoring out commonalities into base classes and then specializing in the derived classes. In this hierarchy, we say that the derived class is a *subclass* of the base class and the base class is a *superclass* of the derived class. These relationships are transitive.

If *X* IS-A *Y*, then *X* is a *subclass* of *Y* and *Y* is a *superclass* of *X*. These relationships are transitive.

The arrows in the hierarchy diagrams reflect the modern convention of pointing toward the top (or *root*) of the hierarchy. The stream hierarchy illustrates some fancier design decisions. Among other things, commonality among `istream` and `ostream` is factored out and placed in `ios`. Also, `iostream` inherits from both `istream` and `ostream`, illustrating multiple inheritance.

The next few sections examine some of the following issues:

- What is the syntax used to derive a new class from an existing base class?
- How does this affect public or private status?
- How do we specialize a method?
- How do we factor out common differences into an abstract class and then create a hierarchy?
- How do we specify that dynamic binding should be used?
- Can we and should we derive a new class from more than one class (*multiple inheritance*)?

4.2 Inheritance Basics

Recall that a derived class inherits all the properties of a base class. It can then add data members, disable functions, alter functions, and add new functions. Each derived class is a completely new class. A typical layout for inheritance is shown in Figure 4.3. C++ tokens are set in boldface. The form of inheritance described here and used almost exclusively throughout the text is *public inheritance*. Note carefully that the word `public` after the colon on line 1 signifies public inheritance. Without it, we have private inheritance, which is not what we want, because only public inheritance models an IS-A relationship. Let us briefly describe a derived class:

Public inheritance models an IS-A relationship.

- Generally all data is private, so we just add additional data members in the derived class by specifying them in the private section.
- Any base class member functions that are not specified in the derived class are inherited unchanged, with the following exceptions: constructor, destructor, copy constructor, and operator=. For those the typical defaults apply, with the inherited portion considered as a member. Thus by default a copy constructor is applied to the inherited portion (considered as a single entity) and then member by member. We will be more specific in Section 4.2.6.
- Any base class member function that is declared in the derived class' private section is disabled in the derived class.[2]
- Any base class member function that is declared in the derived class' public section requires an overriding definition that will be applied to objects of the derived class.
- Additional member functions can be added in the derived class.

The derived class inherits all member functions from the base class. It may accept them, disallow them, or redefine them. Additionally, it can define new functions.

[2] This is bad style, because it violates the IS-A relationship: The derived class can no longer do everything that the base class can.

```
class Derived : public Base
{
    // Any members that are not listed are inherited unchanged
    // except for constructor, destructor,
    // copy constructor, and operator=
  public:
    // Constructors, and destructors if defaults are not good
    // Base members whose definitions are to change in Derived
    // Additional public member functions
  private:
    // Additional data members (generally private)
    // Additional private member functions
    // Base members that should be disabled in Derived
};
```

Figure 4.3 General layout of public inheritance

4.2.1 Visibility Rules

We know that any member that is declared with private visibility is accessible only to methods of the class. Thus any private members in the base class are not accessible to the derived class.

Occasionally we want the derived class to have access to the base class members. There are several options. The first is to use public access. However, public access allows access to other classes in addition to derived classes. We could use a friend declaration, but this is also poor design and would require friend declaration for each derived class.

A *protected class member* is private to every class except a derived class.

If we want to allow access to only derived classes, we can make members protected. A *protected class member* is private to every class except a derived class. Declaring data members as protected or public violates the spirit of encapsulation and information hiding and is generally done only as a matter of programming expediency. Typically, a better alternative is to write accessor and mutator methods. However, if a protected declaration allows you to avoid convoluted code, then it is not unreasonable to use it. In this text, protected data members are used for precisely this reason. Using protected methods is also done in this text. This allows a derived class to inherit an internal method without making it accessible outside the class hierarchy. Figure 4.4 shows the visibility of members in certain situations.

Public inheritance situation	Public	Protecte d	Private
Base class member function accessing *M*	Yes	Yes	Yes

Figure 4.4 Access rules that depend on what *M*'s visibility is in the base class

Public inheritance situation	Public	Protecte d	Private
Derived class member function accessing	Yes	Yes	No
`main`, accessing *B.M*	Yes	No	No
`main`, accessing *D.M*	Yes	No	No
Derived class member function accessing	Yes	No	No
B is an object of the base class; *D* is an object of the publicly derived class; *M* is a member of the base class.			

Figure 4.4 Access rules that depend on what *M*'s visibility is in the base class

4.2.2 The Constructor and Base Class Initialization

Each derived class should define its constructors. If no constructor is written, then a single zero-parameter default constructor is generated. This constructor will call the base class zero-parameter constructor for the inherited portion and then apply the default initialization for any additional data members.

If no constructor is written, then a single zero-parameter default constructor is generated that calls the base class zero-parameter constructor for the inherited portion, and then applies the default initialization for any additional data fields.

Constructing a derived class object by first constructing the inherited portion is standard practice. In fact, it is done by default, even if an explicit derived class constructor is given. This is natural because the encapsulation viewpoint tells us that the inherited portion is a single entity, and the base class constructor tells us how to initialize this single entity.

Base class constructors can be explicitly called by its name in the initializer list. Thus the default constructor for a derived class is in reality

```
public Derived( ) : Base( )
{
}
```

```
class UnderflowException : public underflow_error
{
  public:
    UnderflowException( const string & msg = "" )
      : exception( msg.c_str( ) ) { }
};
```

Figure 4.5 Constructor for new exception class `Underflow`; uses base class initializer list

A base-class initializer is used to call the base class constructor.

The base class initializer can be called with parameters that match a base class constructor. As an example, Figure 4.5 illustrates a class UnderflowException that could be used when implementing data structures. UnderflowException is thrown when an attempt is made to extract from an empty data structure. An UnderflowException object is constructed by providing an optional string. Since the underflow_error class specification requires a primitive string, we need to use an initializer list. The UnderflowException object adds no data members, so the construction method is simply to construct the inherited portion using the underflow_error constructor.

If the base class initializer is not provided, then an automatic call to the base class constructor with no parameters is generated. If there is no such base class constructor, then a compiler error results. Thus, this is a case where initializer lists might be mandatory.

4.2.3 Adding Members

A derived class inherits from its base class the behavior of the base class. This means that all methods defined for the base class are now defined for the derived class. In this section we examine the consequences of adding extra methods and data members.

Our vector class in Section 3.4.2 throws an exception if an out-of-bounds index is detected. It makes no attempt to be fancy, and passes back no information except the fact that an error has occurred. Let us look at an alternative that could have been used (note that exception and <stdexcept> are relatively new language additions, which is why we have elected not to use them in the remainder of the text). The alternative stores information about what went wrong inside the exception object. It provides accessors to get this information. However, it still IS-A exception, meaning that is can be used any place that an exception can be used. The new class is shown in Figure 4.6.

BadIndex has one constructor, and three methods (in addition to defaults for copying and destruction that we ignore for now). The constructor accepts two parameters. It initializes the inherited exception portion using a zero-parameter constructor. It then uses the two parameters to store the index that caused the error and the size of the vector. Presumably, the vector has code such as:

```
// See Figure 3.14
Object & operator[]( int index )
{
    if( index < 0 || index >= currentSize )
        throw BadIndex( index, size( ) );
    return objects[ index ];
}
```

The three methods available for `BadIndex` are `getIndex`, `getSize`, and `what`. The behavior of `what` is unchanged from the `exception` class.

```
// Example of a derived class that adds new members.

class BadIndex : public exception
{
  public:
    BadIndex( int idx, int sz )
      : index( idx ), size( sz ) { }

    int getIndex( ) const
      { return index; }
    int getSize( ) const
      { return size; }

  private:
    int index;
    int size;
};
```

Figure 4.6 `BadIndex` class, derived from `exception`

```
// Use the BadIndex exception.
int main( )
{
    NewVector<int> v( 10 );

    try
    {
        for( int i = 0; i <= v.size( ); i++ )   // off-by-one
            v[ i ] = 0;
    }
    catch( const BadIndex & e )
    {
        cout << e.what( ) << ", index=" << e.getIndex( )
             << ", size=" << e.getSize( ) << endl;
    }

    return 0;
}
```

Figure 4.7 Using the `BadIndex` class

Besides the new functionality, `BadIndex` has two data members in addition to the data members that are inherited from `exception`. What data was inherited from `exception`? The answer is, we do not know (unless we look at the class design), and if the inherited data is private, it is inaccessible. Notice, however, that we do not need this knowledge. Furthermore, our design works regardless of the underlying data representation in `exception`. Thus changes to the private implementation of `exception` will not require any changes to `BadIndex`.

Figure 4.7 shows how the `BadIndex` class could be used. Notice that since a `BadIndex` IS-A `exception`, at line 11 we could catch it using an `exception` reference.[3] We could apply the `what` method to get some information. However, we could not apply the `getIndex` and `getSize` methods, because those methods are not defined for all `exception` objects.

Because the predefined `exception` class is a recent language addition, the online code has a collection of exceptions rooted at class `DSException`.

4.2.4 Overriding a Method

The derived class method must have the same or compatible return type and signature.

Methods in the base class are overridden in the derived class by simply providing a derived class method with the same signature. The derived class method must have the same or compatible return type (the notion of a compatible return type is new, and is discussed in Section 4.4.4.)

Partial overriding involves calling a base class method by using the scope operator.

Sometimes the derived class method wants to invoke the base class method. Typically, this is known as *partial overriding*. That is, we want to do what the base class does, plus a little more, rather than doing something entirely different. Calls to a base class method can be accomplished by using the scope operator. Here is an example:

```
class Workaholic : public Worker
{
  public:
    void doWork( )
    {
        Worker::doWork( );  // Work like a Worker
        drinkCoffee( );     // Take a break
        Worker::doWork( );  // Work like a Worker some more
    }
};
```

[3] Even though the `BadIndex` object is an automatic variable in `operator[]`, it can be caught by reference because thrown objects are guaranteed longer lifetime than normal function arguments.

4.2.5 Static and Dynamic Binding

Figure 4.8 illustrates that there is no problem in declaring Worker and Workaholic objects in the same scope because the compiler can deduce which doWork method to apply. w is a Worker and wh is a Workaholic, so the determination of which doWork is used in the two calls at line 6 is computable at compile time. We call this *static binding* or *static overloading*.

In *static binding,* the decision on which function to use to resolve an overload is made at compile time.

On the other hand, the code in Figure 4.9 is more complicated. If `x` is zero, we use a plain `Worker` class; otherwise, we use a `Workaholic`. Recall that since a `Workaholic` IS-A `Worker`, a `Workaholic` can be accessed by a pointer to a `Worker`. Any method that we might call for `Worker` will have a meaning for `Workaholic` objects. We see then that public inheritance automatically defines a type conversion from a pointer to a derived class to a pointer to the base class. Thus we can declare that wptr is a pointer to the base class `Worker` and then dynamically allocate either a `Worker` or `Workaholic` object for it to point at. When we get to line 9, which doWork gets called?

The decision of which doWork to use can be made at compile time or at run time. If the decision is made at compile time (static binding*)*, then we must use Worker's doWork because that is the type of *wptr at compile time. If wptr is actually pointing at the Workaholic, this is the wrong decision. Because the type of object that wptr is actually pointing at can only be determined once the program has run, this decision must be made at run time. This is known as *dynamic binding*. As we discussed earlier in this chapter, this is almost always the preferred course of action.

If a member function is declared to be virtual, *dynamic binding* is used. The decision on which function to use to resolve an overload is made at run time, if it cannot be determined at compile time.

However a run-time decision incurs some run-time overhead because it requires that the program maintain extra information and that the compiler generate code to perform the test. This overhead was once thought to be significant, and thus although other languages, such as Smalltalk and Objective C, use dynamic binding by default, C++ does not.

```
1     const VectorSize = 20;
2     Worker w;
3     Workaholic wh;
4         ...
5     wh.doWork( )
6     w.doWork( ); wh.doWork( );
```

Figure 4.8 `Worker` and `Workaholic` classes with calls to `doWork` that are done automatically and correctly

In general, if a function is redefined in a derived class, it should be declared virtual in the base class.

Instead, the C++ programmer must ask for it by specifying that the function is virtual. A *virtual function* will use dynamic binding if a compile-time binding decision is impossible to deduce. A non-virtual function will always use static binding. The default, as we implied above, is that functions are non-virtual. This is unfortunate because we now know that the overhead is relatively minor.

Virtualness is inherited, so it can be indicated in the base class. Thus if the base class declares that a function is virtual (in its declaration), then the decision can be made at run time; otherwise, it is made at compile time. For example, in the `exception` class, the `what` method is virtual. The derived classes require no further action to have dynamic binding apply for `what` method calls.

Consequently, for the example in Figure 4.9, the answer depends entirely on whether or not `doWork` was declared virtual in the `Worker` class (or higher in the hierarchy). Note carefully that if `doWork` is not virtual in the `Worker` class (or higher in the hierarchy), but is later made virtual in `Workaholic`, then accesses through pointers and references to `Worker` will still use static binding. To make a run-time decision, we would have to place the keyword virtual at the start of the `doWork` declaration in the Worker class interface (the rest of the class is omitted for brevity):

```
class Worker
{
  public:
    virtual void doWork( );
};
```

```
Worker *wptr;
cin >> x;
if( x != 0 )
    wptr = new Workaholic( );
else
    wptr = new Worker( );

    ...
wptr->doWork( );       // What does this mean?
```

Figure 4.9 `Worker` and `Workaholic` objects accessed though a pointer to a `Worker`; which version of `doWork` is used depends on whether `doWork` is declared virtual in `Worker`

As a general rule, if a function is overridden in a derived class, it should be declared virtual in the base class to ensure that the correct function is selected when a pointer to an object is used. An important exception is discussed in Section 4.2.7.

To summarize: Static binding is used by default, and dynamic binding is used for virtual functions if the binding cannot be resolved at compile time. However, a run-time decision is only needed when an object is accessed through a pointer or reference to a base class.

4.2.6 The Default Constructor, Copy Constructor, Copy Assignment Operator, and Destructor

There are two issues surrounding the default constructor, copy constructor, and copy assignment operator: first, if we do nothing, are these operators private or public? Second, if they are public, what are their semantics?

The public/private status of the default constructor, copy constructor, and copy assignment operator, like all other members is inherited.

We assume public inheritance. We also assume that these functions were public in the base class. What happens if they are completely omitted from the derived class? We know that they will be public, but what will their semantics be? We know that for classes there are defaults for the simple constructor, the copy constructor and the copy assignment operator. Specifically, the default is to apply the appropriate operation to each member in the class. Thus if a copy assignment operator is not specified in a class, we have seen that it is defined as a member-by-member copy. The same rules apply to inherited classes. This means, for instance, that

```
const BadIndex & operator=( const BadIndex & rhs );
```

since it is not explicitly defined, is implemented by a call to operator= for the base class.

If a default destructor, copy constructor, or copy assignment operator is publicly inherited but not defined in the derived class, then by default the operator is applied to each member.

What is true for any member function is in effect true for these operators when it comes to visibility. Thus, if operator= is disabled by being placed in the private section in the base class, then it is still disabled. The same holds true for the copy constructor and default constructor. The reasoning, however, is slightly different. operator= is in effect disabled because a public default operator= is generated. However, by default operator= is applied to the inherited portion and then member by member. Since operator= for the base class is disabled, the first step becomes illegal. Thus placing default constructors, copy constructors, and operator= in the private section of the base class has the effect of disabling them in the derived class (even though technically they are public in the derived class).

4.2.7 Constructors and Destructors: Virtual or not Virtual?

The short answer to the question of whether constructors and destructors should be virtual or not is that constructors are never virtual, and destructors should always be made virtual if they are being used in a base class and should be non-virtual otherwise. Let us explain the reasoning.

Constructors are never virtual.

For constructors a virtual label is meaningless. We can always determine at compile time what we are constructing. For destructors we need virtual to ensure that the destructor for the actual object is called. Otherwise, if the derived class

In an inheritance hierarchy the destructor is always virtual.

consists of some additional members that have dynamically allocated memory, that memory will not be freed by the base class destructor. In a sense the destructor is no different than any other member function. For example, in Figure 4.10 suppose that the base class contains `string`s name1 and name2. Automatically, its destructor will call the destructors for these strings, so we are tempted to accept the default. In the derived class we have an additional `string` newName. Automatically, its destructor calls `newName`'s destructor, and then the base class destructor. So it appears that everything works.

However, if the destructor for the base class is used for an object of the derived class, only those items that are inherited are destroyed. The destructor for the additional data member newName cannot possibly be called because the destructor for the base class is oblivious to newName's existence.

Thus even if the default destructor seems to work, it does not if there is inheritance. The base class constructor should always be made virtual, and if it is a trivial destructor, it should be written anyway, with a virtual declaration and empty body. When the destructor is virtual, we are certain that a runtime decision will be used to choose the destructor that is appropriate to the object being `delete`d.

For a concrete example, Figure 4.11 shows the class interface for `exception`. Notice how the destructor is virtual.

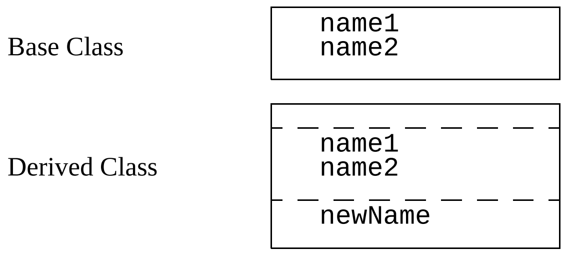

Figure 4.10 Calling the base class destructor does not free memory associated with `newName`

```
// Interface for class exception in <exception>

class exception
{
  public:
    exception( );
    exception( const exception & rhs );

    virtual ~exception( );

    const exception & operator=( const exception & rhs );

    virtual const char * what( );

  private:
    // implementation-defined
};
```

Figure 4.11 Class interface for `exception`

4.2.8 Abstract Methods and Classes

So far we have seen that some methods are invariant over a hierarchy and that other methods can have their meaning changed over the hierarchy. A third possibility is that the method is meaningful for the derived classes and an implementation must be provided for the derived classes; however, that implementation is not meaningful for the base class. In this case, we can declare that the base class method is *abstract.*

An *abstract method* is a method that declares functionality that all derived class objects must eventually implement. In other words, it says what these objects can do. However, it does not provide a default implementation. Instead, each object must provide its own implementation.

An *abstract method* has no meaningful definition and is thus always defined in the derived class.

A class that has at least one abstract method is an *abstract class*. Since the behavior of an abstract class is not completely defined, abstract classes can never be instantiated. When a derived class fails to override an abstract method with an implementation, the method remains abstract in the derived class. As a result, the derived class remains abstract, and the compiler will report an error if an attempt to instantiate the abstract derived class is made.

An example is an abstract class `Shape`, which is used in a larger example later in this chapter. Specific shapes, such as `Circle` and `Rectangle`, are derived from `Shape`. We can then derive a `Square` as a special `Rectangle`. Figure 4.12 shows the class hierarchy that results.

The `Shape` class can have data members that are common to all classes. In a more extensive example, this could include the coordinates of the object's extremities. It declares and provides a definition for methods, such as `positionOf`, that are independent of the actual type of object; `positionOf` would be an invariant method. It also declares methods that apply for each particular type of object. Some of these methods make no sense for the abstract class `Shape`. For instance, it is difficult to compute the area of an abstract object; the `area` method would be an abstract method.

A class with at least one abstract method must be an *abstract class*.

As mentioned earlier, the existence of at least one abstract method makes the base class abstract and disallows creation of it. Thus a `Shape` object cannot itself be created; only the derived objects can. However, as usual, a `Shape` can point to or reference any concrete derived object, such as a `Circle` or `Rectangle`. Thus

```
Shape *a, *b;
a = new Circle( 3.0 );      // Legal
b = new Shape( "circle" );  // Illegal
```

An abstract class object can never be constructed. However, we still provide a constructor that can be called by derived classes.

Figure 4.13 shows the abstract class `Shape`. At line 30, we declare a `string` that stores the type of shape. This is used only for the derived classes. The member is private, so the derived classes do not have direct access to it. The rest of the class specifies a collection of methods.

The constructor never actually gets called directly because `Shape` is an abstract class. We need a constructor, however, so that the derived class can call it to initialize the private members. The `Shape` constructor sets the internal `name` data member. Notice the virtual destructor, in according with the discussion in Section 4.2.7.

Abstract methods are also known as *pure virtual functions* in C++.

Line 21 of Figure 4.13 declares the abstract method `area`. A method is declared abstract by specifying that it is `virtual`, and supplying `= 0` in the interface in place of an implementation. Because of the syntax, abstract methods are also known as *pure virtual functions* in C++. As with all virtual methods, a run-time decision will select the appropriate `area` in a derived class. `area` is an abstract method because there is no meaningful default that could be specified to apply for an inherited class that chose not to define its own.

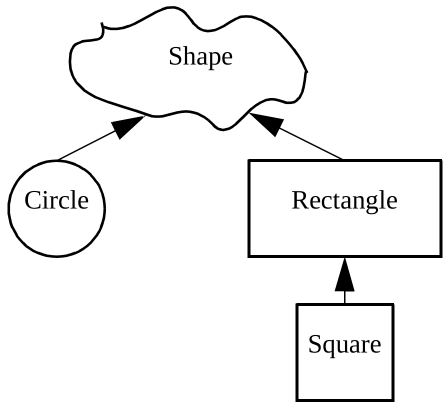

Figure 4.12 The hierarchy of shapes used in an inheritance example

```
 1 // Abstract base class for shapes
 2 //
 3 // CONSTRUCTION: is not allowed; Shape is abstract
 4 //
 5 // ******************PUBLIC OPERATIONS*********************
 6 // double area( )           --> Return the area (abstract)
 7 // bool operator< ( rhs )   --> Compare 2 Shape objects by area
 8 // void print( out = cout)  --> Standard print method
 9
10 #include <iostream>
11 #include <string>
12 using namespace std;
13
14 class Shape
15 {
16   public:
17     Shape( const string & shapeName = "" ) : name( shapeName )
18       { }
19     virtual ~Shape( ) { }
20
21     virtual double area( ) const = 0;
22
23     bool operator< ( const Shape & rhs ) const
24       { return area( ) < rhs.area( ); }
25
26     virtual void print( ostream & out ) const
27       { out << name << " of area " << area( ); }
28
29   private:
30     string name;
31 };
```

Figure 4.13 Abstract base class `Shape`

The comparison method shown at lines 23 to 24 is not abstract because it can be meaningfully applied for all derived classes. In fact, its definition is invariant throughout the hierarchy of shapes, so we have not made it virtual.

The `print` method, shown at lines 26 and 27, prints out the name of the shape and its area. Although it appears to be invariant now, we make it virtual just in case we change our mind later on. `operator<<` is written in Figure 4.14.

```
// Output routine for Shape
ostream & operator<< ( ostream & out, const Shape & rhs )
{
    rhs.print( out );
    return out;
}
```

Figure 4.14 Output routine for `Shape` that includes its name and area

Static binding is used for a nonvirtual function when the function is invariant over the inheritance hierarchy.

Before continuing, let us summarize the three types of member functions:

1. *Nonvirtual functions*. Overloading is resolved at compile time. To ensure consistency when pointers to objects are used, we generally use a nonvirtual method only when the function is invariant over the inheritance hierarchy (that is, when the method is never redefined).The exception to this rule is that constructors are always nonvirtual, as mentioned in Section 4.2.7.
2. *Virtual functions*. Overloading is resolved at run time. The base class provides a default implementation that may be overridden by the derived classes. Destructors should be virtual, as mentioned in Section 4.2.7.
3. *Pure virtual functions*. Overloading is resolved at run time. The base class provides no implementation and is abstract. The absence of a default requires either that the derived classes provide an implementation or that the derived classes themselves be abstract.

4.3 Example: Expanding the `Shape` Class

This section implements the derived `Shape` classes and shows how they are used in a polymorphic manner. The following problem is used:

SORTING SHAPES

Read N shapes (circles, squares, or rectangles) and output them sorted by area.

The implementation of the derived classes, shown in Figure 4.15, is completely straightforward and illustrates almost nothing that we have not already seen. The only new item is that `Square` is derived from `Rectangle`, which itself is derived from `Shape`. This derivation is done exactly like all the others. In implementing these classes, we must do the following:

1. Provide a new constructor.
2. Examine each virtual function to decide if we are willing to accept its defaults. For each virtual function whose defaults we do not like, we must write a new definition.
3. Write a definition for each pure virtual function.
4. Write additional member functions if appropriate.

For each class, we provide a simple constructor that allows initialization with basic dimensions (radius for circles, side lengths for rectangles and squares). We first initialize the inherited portion by calling the base class initializer. Each class is required to provide an `area` method because `Shape` has declared that it is an abstract method. If the `area` method is not provided for some class then an error will be detected at compile time. This is because if an implementation of `area` is missing, a derived class will itself be abstract. Note that `Square` is willing to inherit the `area` method from the `Rectangle`, so it does not provide a redefinition. Note also that its name internally is now a rectangle.

```
// Circle, Square, Rectangle class interfaces;
//     all based on Shape
//
// CONSTRUCTION: with (a) no initializer or (b) radius (for
//     circle), side length (for square), length and width
//     (for rectangle)
// ******************PUBLIC OPERATIONS**********************
// double area( )       --> Implements Shape pure virtual area

const double PI = 3.1415927;

class Circle : public Shape
{
  public:
    Circle( double rad = 0.0 )
      : Shape( "circle" ), radius( rad ) { }
    double area( ) const
      { return PI * radius * radius; }

  private:
    double radius;
};

class Rectangle : public Shape
{
  public:
    Rectangle( double len = 0.0, double wid = 0.0 )
      : Shape( "rectangle" ), length( len ), width( wid ) { }
    double area( ) const
      { return length * width; }

  private:
    double length;
    double width;
};

class Square : public Rectangle
{
  public:
    Square( double side = 0.0 )
      : Rectangle( side, side ) { }
};
```

Figure 4.15 Complete `Circle`, `Rectangle`, and `Square` classes

We can only declare arrays of pointers to base classes because the size of the base class is usually smaller than the size of the derived class. It can never be larger.

Now that we have written the classes, we are ready to solve the original problem. What we would like to do is declare an array of Shapes. But we cannot declare one Shape, much less an array of them. There are two reasons for this. First, Shape is an abstract base class, so a Shape object does not exist. Even if Shape was not abstract, which would be the case if it defined an area function, we still could not reasonably declare an array of Shapes. This is because the basic Shape has one data member, Circle adds a second data member, Rectangle adds a third data member, and so on. The basic Shape is not large enough to hold all of the possible derived types. Consequently, we need an array of pointers to Shape. Figure 4.16 attempts this approach; however, it does not quite work because we get in trouble at the sorting stage.

We examine the logic in Figure 4.16 and show how to correct the deficiency. First we read the objects. At line 17 we are actually reading a character and then the dimensions of some shape, creating a shape, and finally assigning a pointer to point at the newly created shape. Figure 4.17 shows a bare bones implementation. So far so good.

```
 1 #include <iostream>
 2 #include <vector>
 3 using namespace std;
 4
 5 // main: read shapes and output increasing order of area.
 6 // Error checks omitted for brevity.
 7 int main( )
 8 {
 9     int numShapes;
10     cin >> numShapes;
11     vector<Shape *> array( numShapes );  // Array of Shape *
12
13       // Read the shapes
14     for( int i = 0; i < numShapes; i++ )
15     {
16         cout << "Enter a shape: ";
17         cin >> array[ i ];
18     }
19
20     insertionSort( array );
21
22     cout << "Sorted by increasing size:" << endl;
23     for( int j = 0; j < numShapes; j++ )
24         cout << *array[ j ] << endl;
25
26     return 0;
27 }
```

Figure 4.16 `main` routine to read shapes and output them in increasing order of area

```
// Create an appropriate Shape object based on input.
// The user types 'c', 's', or 'r' to indicate the shape
// and then provides dimensions when prompted.
// A zero-radius circle is returned for any error.
istream & operator>>( istream & in, Shape * & s )
{
    char ch;
    double d1, d2;

    in.get( ch );      // First character represents shape
    switch( ch )
    {
      case 'c':
        in >> d1;
        s = new Circle( d1 );
        break;

      case 'r':
        in >> d1 >> d2;
        s = new Rectangle( d1, d2 );
        break;

      case 's':
        in >> d1;
        s = new Square( d1 );
        break;

      case '\n':
        return in >> s;

      default:
        cerr << "Needed one of c, r, or s" << endl;
        s = new Circle;      // Radius is 0
        break;
    }

    return in;
}
```

Figure 4.17 Simple input routine for reading a pointer to a `Shape`

We then call insertionSort to sort the shapes. Recall that we already have a insertionSort template from Section 3.3. Since array is an array of pointers to shapes, we expect that it will work as long as we provide a comparison routine with the declaration

```
int operator<( const Shape * lhs, const Shape * rhs );
```

If a class is instantiated with pointer types, shallow operations are used.

Unfortunately, that does not work. insertionSort uses the `operator<` that already exists for pointers. That operator compares the addresses being pointed at, which guarantees that the array will be unaltered (because a[i] is always stored at a lower address than a[j] if i<j).

```
struct PtrToShape
{
    Shape *ptr;

    bool operator< ( const PtrToShape & rhs ) const
      { return *ptr < *rhs.ptr; }

    const Shape & operator*( ) const
      { return *ptr; }
};

// main: read shapes and output increasing order of area.
// Error checks omitted for brevity.
int main( )
{
    int numShapes;
    cout << "Enter number of shapes: ";
    cin >> numShapes;

      // Read the shapes
    vector<PtrToShape> array( numShapes );

    for( int i = 0; i < numShapes; i++ )
    {
        cout << "Enter a shape (c, r, or s with dimensions): ";
        cin >> array[ i ].ptr;
    }

    insertionSort( array );
    cout << "Sorted by increasing size:" << endl;
    for( int j = 0; j < numShapes; j++ )
        cout << *array[ j ] << endl;

    for( int k = 0; k < numShapes; k++ )
        delete array[ k ].ptr;

    return 0;
}
```

Figure 4.18 `main` routine reads shapes and outputs them in increasing order of area

To make this work, we need to define a new class that hides the fact that the objects we are storing and sorting are pointers. This is shown in Figure 4.18. The PtrToShape object stores the pointer to a Shape and provides a comparison function that compares `Shape`s rathers than pointers. It does this by dereferencing both pointers and calling the `Shape operator<` on the resulting `Shape` objects. Note that we make excessive calculations to compute areas. Avoiding this is left as Exercise 4.13. Note also that in general, we must call `delete` to reclaim the memory consumed by the `Shape` objects.

Deep comparison semantics can be obtained by designing a class to store the pointer.

The `PtrToShape` class also overloads the unary * operator, so that a PtrToShape object looks just like a pointer to a Shape. We certainly can add more members to hide information better, but we prefer to keep things as short as possible. The idea of wrapping a pointer inside a class is a common design pattern. We look at this recurring theme in Section 5.3.

4.4 Tricky C++ Details

Inheritance in C++ has numerous subtle points. Some of these are discussed in this section.

4.4.1 Static Binding of Parameters

Dynamic binding means that the member function that is appropriate for the object being operated on is used. However, it does not mean that the absolute best match is performed for all parameters. Specifically, in C++, the parameters to a method are always deduced statically, at compile time.

In C++, the parameters to a method are always deduced statically, at compile time.

For a concrete example, consider the code in Figure 4.19. In the `whichFoo` method, a call is made to `foo`. But which `foo` is called? We expect the answer to depend on the runtime types of `arg1` and `arg2`.

Because parameters are always matched at compile time, it does not matter what type `arg2` is actually referencing. The `foo` that is matched will be

```
    virtual void foo( const Base & x );   // METHOD A or C
```

The only issue is whether the `Base` or `Derived` version is used That is the decision that is made at runtime, when the object that `arg1` references is known.

Static binding has important ramifications. Consider the following situation in which we overload the output operator for both a base class and derived class.

```
ostream & operator<< ( ostream & out, const Base & x );
ostream & operator<< ( ostream & out, const Derived & x );
```

Suppose we now try to call the output function.

```
    Base *b = new Derived;
    cout << *b << endl;
```

Because parameters are statically deduced, output is done (unfortunately) using the `operator<<` that takes a `Base` parameter.

```
class Derived;   // Incomplete declaration

class Base
{
  public:
    virtual void foo( const Base & x );      // METHOD A
    virtual void foo( const Derived & x );   // METHOD B
};

class Derived : public Base
{
  public:
    virtual void foo( const Base & x );      // METHOD C
    virtual void foo( const Derived & x );   // METHOD D
};

void whichFoo( Base & arg1, Base & arg2 )
{
    arg1.foo( arg2 );
}
```

Figure 4.19 Illustration of static binding for parameters

However, recall that we have been recommending the approach of having the class define a `print` method, and then implementing `operator<<` by calling the `print` method. If we do this, we only need to write `operator<<` for the base class:

```
ostream & operator<< ( ostream & out, const Base & x )
{
    out.print( x );    // print is deduced at run time
    return out;
}
```

Now the base class and derived class each provide their own version of the `print` method. `operator<<` is called for all `Base` and `Derived` objects. However, when that happens, the call to `print` uses dynamic binding!

4.4.2 Default Parameters

Default parameters are statically bound, meaning that they are deduced at compile time. It is unsafe to change the default value in a derived class because this can create an inconsistency with virtual functions, which are bound at run time.

It is unsafe to change the default value in a derived class.

4.4.3 Derived Class Methods Hide Base Class Methods

C++ has an annoying feature illustrated by the example in Figure 4.20. In the code, we have a base class and a derived class. The base class declares a function named `bar`, with zero parameters. The derived class adds a function named `bar`, with one parameter.

In `test`, we illustrate the various calls that can be made. At line 15, we attempt to call the zero-parameter `bar` through a `Base` reference. We expect this to work and it does. Notice that the actual object being acted upon could be a `Derived` object. The next line attempts to call the one-parameter `bar` through a `Base` reference. Since this is not defined for `Base` objects, it must fail, and indeed, the line does not compile. The one-parameter `bar` must be called through a `Derived` reference, as shown on line 17.

So far all is good. Now comes the unexpected part. If we call the zero-parameter `bar` with a `Derived` reference, the code does not compile. This is unexpected, since the code at line 15 compiles, and a `Derived` IS-A `Base`.

What has happened appears to be a language flaw. When a method is declared in a derived class, it hides all methods of the same name in the base class. Thus `bar` is no longer accessible through a `Derived` reference, even though it would be accessible through a `Base` reference:

```
Base & tmp = arg3; tmp.bar( );   // Legal!
```

There are two ways around this. Once way is to override the zero-parameter `bar` in `Derived`, with an implementation that calls the `Base` class version. In other words, in `Derived`, add:

```
void bar( ) { Base::bar( ); }   // In class Derived
```

The other method is newer and does not work on all compilers. Introduce the base class member function into the derived class scope with a `using` declaration:

```
class Base
{
  public:
    virtual void bar( );           // METHOD A
};

class Derived : public Base
{
  public:
    virtual void bar( int x );     // METHOD B
};

void test( Base & arg1, Derived & arg2, Derived & arg3 )
{
    arg1.bar( );      // Compiles, as expected.
    arg1.bar( 4 );    // Does not compile, as expected.
    arg2.bar( 4 );    // Compiles, as expected.
    arg3.bar( );      // Does not compile. Not expected.
}
```

Figure 4.20 Illustration of hiding

```
    using Base::bar;                 // In class Derived
```

The most important reason you should be aware of this rule is that many compilers will issue a warning when you hide a member function. Since a signature includes whether or not a function is an accessor, if the base class function is an accessor (a constant member function), and the derived class function is not, you have usually made an error, and this is how the compiler might let you know about it. Pay attention to these warnings.

4.4.4 Compatible Return Types for Overridden Methods

Return types present an important difficulty. Consider the following operator, defined in a base class:

```
virtual const Base & operator++( );
```

The derived class inherits it,

```
const Base & operator++( );
```

but that is not really what we want. If we have a return type in the derived class, it ought to be a constant reference to the derived type and not the base type. Thus the operator++ that is inherited is not the one we want. We would like instead to override `operator++` with:

```
const Derived & operator++( );
```

Recall that overriding a function means writing a new function with the same signature. Under original C++ rules, the return type of the new and overridden function had to match exactly.

Under the new rules, the return type may be relaxed. By this we mean that if the original return type is a pointer (or reference) to *B*, the new return type may be a pointer (or reference) to *D*, provided *D* is a publicly derived class of *B*. This corresponds to our normal expectation of IS-A relationships.

If the original return type is a pointer (or reference) to *B*, the new return type may be a pointer (or reference) to *D*, provided *D* is a publicly derived class of *B*.

4.4.5 Private Inheritance

Private inheritance means that even public members of the base class are hidden. Seems like a silly idea, doesn't it? In fact it is, if we are talking about implementing an IS-A relationship. Private inheritance is thus generally used to implement a HAS-A relationship (that is, a derived class *D* has or uses a base class *B*).

Private inheritance means that even public members of the base class are hidden.

In many cases we can get by without using inheritance: We can make an object of class *B* a member of class *D* and, if necessary, make *D* a friend of *B*. This is known as *composition*. Composition is the preferred mechanism, but occasionally private inheritance is more expedient or slightly faster (because it avoids a layer of function calls). For the most part, it is best to avoid private inheritance unless it greatly simplifies some coding logic or can be justified on performance grounds. However, in Section 5.3.3, we will see an appropriate and typical use of private inheritance.

Composition is preferred to private inheritance. In composition, we say that class *B* is composed of class *A* (and other objects).

It is important to remember that by default, *private inheritance* is used. If the keyword `public` was omitted on line 3 of Figure 4.6, we would have private inheritance. In that case the public member functions of `exception` would still be inherited, but they would be private members of `BadIndex` and they could not be called by users of `BadIndex`. Thus the `what` method would not be visible. The type compatibility of base class and derived class pointers and references described earlier does not apply for nonpublic inheritance. Thus, in the following code, a `BadIndex` exception would not be caught:

The default is private inheritance but it should be avoided.

```
catch( const exception & e ) { cout << e.what( ) << endl; }
```

4.4.6 Friends

Are friends of a class still friends in a derived class? The answer is no. For example, suppose *F* is a friend of class *B*, and *D* is derived from *B*. Suppose *D* has nonpublic member *M*. Then in class *D*, *F* does not have access to *M*. However, the inherited portion of *B* is accessible to *F* in class *D*. Figure 4.21 summarizes the results. *D* can declare that *F* is also a friend, in which case all of *D*'s members would be visible.

Friendship is not inherited.

4.4.7 Call by Value and Polymorphism Do Not Mix

Consider the following statement, assume that `BadIndex` is publicly inherited from `exception`, and suppose that it has overridden the `what` method:

```
catch( exception e ) { cout << e.what( ) << endl; }
```

Notice that `e` is passed using call by value. Now suppose a `BadIndex` exception has been thrown. Which `what` method gets called? The answer is not what we want.

Public inheritance situation	Public	Protected	Private
F accessing *B.MB*	Yes	Yes	Yes
F accessing *D.MD*	Yes	No	No
F accessing *D.MB*	Yes	Yes	Yes

B is an object of the base class; *D* is an object of the publicly derived class; *MB* is a member of the base class. *MD* is a member of the derived class. *F* is a friend of the base class (but not the derived class)

Figure 4.21 Friendship is not inherited

Slicing is the loss of inherited data members when a derived class object is copied into a base class object.

When we use call by value, the actual argument is always copied into the formal parameter. This means that the `BadIndex` object is copied into `e`. This is done by using `e`'s `operator=`, which means that only the `exception` component of `BadIndex` is copied. (This is known as slicing.) In any event, the type of `e` is `exception`, and so it is the `exception` class' `what` method that is called, and it is acting on a trimmed portion of the `BadIndex` object. The moral of the story: polymorphism and call by value do not mix.

4.5 Multiple Inheritance

Multiple inheritance is used to derive a class from several base classes. We do not use multiple inheritance in this book.

All the inheritance examples seen so far derived one class from a single base class. In *multiple inheritance* a class may be derived from more than one base class. As an example, in the iostream library, an iostream (which allows both reading and writing) is derived from both an istream and an ostream. As a second example, a university has several classes of people, including: students and employees. But some people are both students and employees. The StudentEmployee class could be derived from both the Student class and the Employee class; each of those classes could be derived from the abstract base class UniversityPerson.

In multiple inheritance the new class inherits members from all of its base classes. This leads to some immediate problems that the user will need to watch out for:

- Suppose UniversityPerson has class members name and ssn. Then these are inherited by Student and Employee. However, since StudentEmployee inherits the data members from both Student and Employee, we will get two copies of name and ssn unless we use *virtual inheritance*, as in the following:

```
class Student : virtual public UniversityPerson {...}
class Employee :virtual public UniversityPerson {...}
class StudentEmployee : public Student,
                        public Employee {...}
```

- What if Student and Employee have member functions that are augmented to Employee but have the same signatures? For instance, the credit function, not given in UniversityPerson, is added to Student to mean the number of credits for which a student is currently registered. For employees the function returns the number of vacation days still left. Consider the following:

```
UniversityPerson *p = new StudentEmployee;
cout << p->Student::credits( );    // OK
cout << p->Employee::credits( );   // OK
cout << p->credits( );             // Ambiguous
```

- Suppose UniversityPerson defines a virtual member function f, and Student redefines it. However, Employee and StudentEmployee do nothing. Then, for p defined in the previous example, is p->f() ambiguous? In the example above, the answer is no because UniversityPerson is a virtual base class with respect to Student; consequently, Student::f( ) is said to *dominate* UniversityPerson::f( ), and there is no ambiguity. There would be an ambiguity if we did not use virtual inheritance for Student.

Does all this make your head spin? Most of these problems tend to suggest that multiple inheritance is a tricky feature that requires careful analysis before use. Generally speaking, multiple inheritance is not needed nearly as often as we might suspect, but when it is needed it is extremely important. Although the rules for multiple inheritance are carefully defined in the language standard, it is also an unfortunate fact that many compilers have bugs associated with this feature (especially in conjunction with others).

We will not use multiple inheritance in this text. The most common (and safe) way to use multiple inheritance is to inherit only from classes that define no data members and no implementations. Such classes specify protocols only, and most of the ambiguity problems described above go away. A popular programming language, Java, formalizes this into a special class called, interestingly enough, the *interface*. Java does not allow arbitrary multiple inheritance, but does allow multiple interfaces, and the result seems to be very clean code.

You should avoid general use of multiple inheritance in C++ until you are extremely comfortable with simple inheritance and virtual functions; many object-oriented languages (such as Smalltalk, Object Pascal, Objective C, and Ada) do not support multiple inheritance, so you can live without it.

Summary

Inheritance is a powerful feature that allows the reuse of code. However, make sure that functions applied to objects created at run time through the new operator are bound at run time. This feature is known as dynamic binding, and the use of virtual functions is required to ensure that run-time decisions are made. Reread this chapter as often as necessary to make sure you understand the distinction between nonvirtual functions (in which the same definition applies throughout the inheritance hierarchy, and thus compile-time decisions are correct), virtual functions (in which the default provided in the base class can be overwritten in the derived class; run-time decisions are made if needed), and pure virtual functions (which have no default definition).

In this chapter we also saw the programming techniques of wrapping a variable inside a class (Figure 4.18), and mentioned the occasional usefulness of private inheritance. These are two examples of design patterns: techniques that we see over and over again. The next chapter discusses some common design patterns.

Objects of the Game

abstract base class A class with at least one pure virtual function. (184)

abstract method A method that has no meaningful definition and is thus always defined in the derived class. (183)

base class The class on which the inheritance is based. (172)

composition Preferred mechanism to private inheritance when an IS-A relationship does not hold. In composition, we say that an object of class *B* is composed of an object of class *A* (and other objects). (195)

derived class A completely new class that nonetheless has some compatibility with the class from which it was derived. (172)

dynamic binding A run-time decision to apply the method corresponding to the actual referenced object. Used when a member function is declared to be virtual and the correct method cannot be determined at compile time. (179)

HAS-A relationship A relationship in which the derived class has a (property of the) base class. (169)

inheritance The process whereby we may derive a class from a base class without disturbing the implementation of the base class. Also allows the design of class hierarchies, such as `exception`. (172)

IS-A relationship A relationshup in which the derived class is a (variation of the) base class. (169)

multiple inheritance The process of deriving a class from several base classes. (196)

nonvirtual functions Used when the function is invariant over the inheritance hierarchy. Static binding is used for nonvirtual functions. (186)

partial overriding The act of augmenting a base class method to perform additional, but not entirely different, tasks. (178)

polymorphism The ability of a reference or pointer variable to reference or point to objects of several different types. When operations are applied to the variable, the operation that is appropriate to the actual referenced object is automatically selected. (171)

private inheritance The process occasionally used to implement a HAS-A relationship. Even public members of the base class are hidden. (195)

protected class member Accessible by the derived class but private to everyone else. (174)

public inheritance The process by which all public members of the base class remain public in the derived class. Public inheritance models an IS-A relationship. (173)

pure virtual function An abstract method. (184)

slicing The loss of inherited data when a derived class object is copied into a base class object. (196)

static binding/overloading The decision on which function to use is made at compile time. (179)

virtual functions A function for which dynamic binding is used. It should be used if the function is redefined in the inheritance hierarchy. (179)

Common Errors

1. Inheritance is private by default. A common error is to omit the keyword public that is needed to specify public inheritance.
2. If a base class member function is redefined in a derived class, it should be made virtual. Otherwise, the wrong function could be called when accessed through a pointer or reference.
3. Base class destructors should be declared as virtual functions. Otherwise, the wrong destructor may get called in some cases.
4. Constructors should never be declared virtual.
5. Objects of an abstract base class cannot be instantiated.

6. If the derived class fails to implement any inherited pure virtual function, then the derived class becomes abstract and cannot be instantiated, even if it makes no attempts to use the undefined pure virtual function.
7. Never redefine a default parameter for a virtual function. Default parameters are bound at compile time, and this can create an inconsistency with virtual functions that are bound at run time.
8. To access a base class member, the scope resolution must be used. Otherwise, the scope is the current class.
9. Friendship is not inherited.
10. In a derived class, the inherited base class members can only be initialized as an aggregate in a constructor's initializer list. If these members are public or protected, they may later be read or assigned to individually.
11. A common error is to declare a virtual destructor in an abstract base class but not provide an implementation (virtual~Base() or virtual~Base()=0). Both are wrong because the derived class destructor needs to call the base class destructor. If there is nothing to do, then use {} as the definition.
12. If a constructor declaration is provided in the base class, provide the definition, too, for the same reason that we saw in the destructor case.
13. The return type in a derived class cannot be redefined to be different from the base class unless they are both pointer or both reference types, and the new return type is type-compatible with the original.
14. If the base class has a constant member function *F* and the derived class attempts to define a nonconstant member function *F* with an otherwise identical signature, the compiler will warn that the derived *F* hides the base *F*. Heed the warning and find a workaround.

On the Internet

Three self-contained files plus a set of exception classes are available..

Except.h	Contains the exception hierarchy.
Shape.cpp	The Shape example.
StaticBinding.cpp	Contains the code in Figure 4.19 illustrating that parameters are statically bound.
Hiding.cpp	Contains the code in Figure 4.20 illustrating how methods are hidden.