Chapter 12: Derived Classes

A consequence of inheritance:

> **The sins of the father are to be laid upon the children.**
> . . . Euripides, Exodus, and Shakespeare.
> . . . And so are the strengths and skills of the father, in C++.

12.1 How Derivation is Used

Purposes. A class may be derived from another class for several possible reasons:

1. To add functions to those defined by an existing class.

2. To extend and specialize the actions of a function defined in an existing class.

3. To mask a function in an existing class and prevent further access to it.

4. To create a different interface for an existing class.

5. To add data members to those included by an existing class.

6. To further restrict the protection level of data members that belong to an existing class.

Declaration syntax.

1. The first line of the class declaration declares the derivation relationship. The following two lines are taken from the demo program below.

   ```
   class Printed : private Pubs { ... };
   class Book : public Printed { ... };
   ```

2. The style of derivation (public or private) will be clarified later. It affects future derivation steps but not the current derived class.

3. The base class must be defined first; in this case, it is Pubs. The second class (Printed) was derived from Pubs using private derivation. The third class, Book, was derived from Printed by public derivation.

Usage patterns and rules.

1. Typically, more than one class is derived from a base class, but the derived classes form a bush, not a chain.

2. Inheritance chains such as `Pubs -◁-- Printed -◁-- Book` do occur in real programs but are not the most important use of derivation.

3. Two or more derived classes at the same level (brothers) are used to implement polymorphic types and/or multiple interfaces for the same type.

4. It is also possible to derive one class from two or more printed classes. When that is done, an object of each base class is a part of the derived class.

UML for derived classes. To diagram a derived class, we use a line and a triangle with its point toward the printed class and its flat side toward the derived class. The sign written on the lines (`+`, `#`, or `-`) denotes public, protected, or private derivation, respectively. Figure 12.1 gives a UML diagram for the Publications demo program, below. (Note: the Online and Periodical classes are not implemented.)
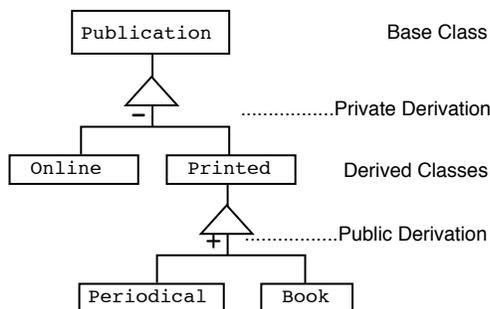
Figure 12.1: UML diagram for private and public inheritance.

12.1.1 Resolving Ambiguous Names

It is normal for a base class and a derived class to have methods with the same name (purposes 2 and 3, above). In this case, we say that the method in the derived class *overrides* the base method. This relationship is the basis of polymorphism (covered in a later chapter). Most of the time, it causes no confusion to have two methods (or data members) in the same class hierarchy that share the same name. When a function method in class A refers to a class member named `mem`, the instance of `mem` in class A will be used if there is one. Otherwise, the compiler will look for an inherited instance of `mem`. (It will look in A's base class and continue searching up the hierarchy until a definition of `mem` is found.)

Sometimes, however, a class may contain a definition for `mem` but one of its methods must refer to a different instance of `mem` defined in the base class. For example, suppose you wish to print the data for a Book, and class Book is derived from class Printed. To print all the Book data, we first tell the base class to print all its data, then we output the local data. To do this, we must be able to call the print method in the base class. This is the purpose of the *scope-resolution operator*, which is written with two colons (`::`).

In the demo program that follows, all three classes have members named `serial`, `name`, and print(), so these names are ambiguous. Line 69 shows the `::` operator used to call the base-class `Pubs::print()` from within the print method of the derived class, `Printed::print()`. On line 93 of Book::print, we write `<<name` to print Book::name. On the next line, we want to print the name from the base class, so we write `<<Printed::name`.

```
93      out <<name <<" #" <<serial
94          <<"\n\tis " <<Printed::name <<endl
95          <<"\tmy Pubs name is hidden from me." <<endl;
```

On line 100 we want to print Pubs::name, but this member is not visible to the Book functions because of the private derivation step even though this member is public for the world at large (a strange combination). In earlier versions of standard C++, the Pubs data could be accessed by using a relabeling cast to convert `this` to a Pubs* and use the result to access the name. In the current ISO standard C++, even this cast is prohibited, and there seems to be *no way* that a function in the Book class can access data in the Pubs class, even when that data is public.

12.1.2 Ctor Initializers

The difference between initialization and assignment is important in both C and C++ because the rules for initialization are more liberal than the rules for assignment. For example, you can and must initialize a const variable but you cannot assign a new value to it.

In C++, the differences are even more important because operator= and the copy constructor may have different definitions. Thus, ctor initializers are a necessary part of C++, and we must be able to use ctors to initial the base-class portion of an object in a derived class. Construction and initialization happen in this order:

- Space on the run-time stack is allocated for the core portion of the new object. This includes space for the core portion of all inherited and new data members of the class. Base class members come first in this object, followed by members declared in the derived classes, in order.

Once the core space exists, the extensions must be created and both core portions and extensions must be initialized. These tasks are always done in the same order, starting at the first data member and progressing downward (in the source code) or to higher addresses (in memory).

- If the object is of a derived class, the members of the base class must be initialized first, since other parts of the object might refer to them. To do this, one of the constructors for the base-class is selected and run. It *initializes* the core portion of the base-class object (which has already been allocated). In the process, it constructs the extensions for this part of the new object.

- If the constructor for the base class requires parameters, they must be supplied by a ctor of this form:

      BaseClassName(argument-list)

  If the base class has more than one constructor, the compiler will select the one whose parameter list matches the list of agruments given by the ctor. If no ctor is given for the base class, the compiler will use the default constructor, if it exists.

  If an object (like a Book object) has a base class (such as Printed) that is also a derived class, the constructor for the middle-level class must pass on parameters to the constructor of the original base class.

- Next, the remaining ctor initializers are used to initialize the data members of the object. These ctors have the form:

      member_name(initial_value)

- Finally, the code portion of the object constructor is run. This code can do anything. It is usually used to allocate the extension portion of the object, set various fields to 0, and connect pointers into a legal and meaningful data structure.

12.2   Visibility and Protection Level

Protected members.   Previously, we have used just two protection levels: *private* and *public*. A third level, *protected*, is intermediate between these two. A public member can be read or written by any part of the program in which its name is known. A private member can be used only by the functions in the same class. A protected class member can also be used by functions of any derived class. In "family" terminology, a Printed keeps his bedroom private, shares protected resources (the home) with his Bookren and his grandBookren but not with strangers, and may provide some resources (such as a sidewalk) for public use.

Public and private derivation.   Members of a base class can be declared as private, protected, or public. During a derivation step, the protection level can be kept the same (by using public derivation) or tightened up (by using private or protected derivation). If public derivation is used, inherited members have the same protection in the derived class as in the base class. With protected derivation, public members become protected in the derived class. With private derivation, all inherited members become private.

The chart in Figure 12.3 summarizes the effects of each of the possible protection combinations. In it, the first column lists members of the Pubs class with protection levels, the second lists the three ways the Printed class can be derived from the Pubs class. In all cases, the Book class is derived publicly from the Printed. The fourth and sixth columns show which members of the Pubs class are visible in the derived class after the first and second derivation steps. From the chart you can see:

- The *protection level* of a member in a derived class (Printed or Book) is the maximum of the protection level in the base class (Pubs) and the styles of all following derivation steps.

- *Accessibility* in the second (Printed) is determined by the *protection level* of a member in the first class (Pubs). The functions of the derived Printed class can access the public and protected members that have been inherited, but not the private members. The inherited private members are there and take up space in a Printed object, but they are "invisible" to the functions of the Printed class. To use such members, a Printed class function would call a public or protected Pubs function.

- *Accessibility* in the third class, (Book), is determined by the *style* of the first derivation (Printed:Pubs). For example, suppose we tried to write the following code in Book::print():

```
cout <<"\n\tMy Pubs is " << Pubs::name<<'\n';
```

  If Printed is derived privately from Pubs, this code generates a compile-time protection error: "member 'Pubs::name' is private in this context". However, the same code in Printed::print() is legal because in the context of the Printed class, name is public.

  In earlier versions of standard C++, the Pubs data could be accessed by using a relabeling cast to convert `this` to a Pubs* and use the result to access the name:

```
out    <<" my Pubs is " <<((Pubs*)this)->name <<endl;
```

  In the current ISO standard C++, even this cast is prohibited, and there seems to be *no way* that a function in the Book class can access data in the Pubs class, even when that data is public.

  In contrast, if protected derivation is used to derive Printed from Pubs, the same code is legal in the Book class. It is OK because Pubs::name is protected in the Printed class and, therefore, visible within Book.

Original protection level in Pubs	Style of 1st Derivation	After 1st derivation		After 2nd derivation	
		Accessible in Printed	Protection in Printed	Accessible in Book	Protection in Book
Grands (private)	public	No	private	No	private
serial (protected)		Yes	protected	Yes	protected
name (public)		Yes	public	Yes	public
Grands (private)	protected	No	private	No	private
serial (protected)		Yes	protected	Yes	protected
name (public)		Yes	protected	Yes	protected
Grands (private)	private	No	private	No	private
serial (protected)		Yes	private	No	private
name (public)		Yes	private	No	private

Figure 12.3: How derivation affects protection level.

12.2.1   Inherited Functions

Functions (as well as data members) are inherited, and the protection rules for inherited functions are the same as for inherited data members. Inherited functions play an important role in maintaining privacy: they enable an object to use its inherited private parts that otherwise would be "invisible" to objects in the derived class. The derived class can use the inherited function without modification or redefine it. A redefinition can take the form of an extension or an override.

Function redefinition.   The functions of a class are closely tied to the representation of the class. Also, certain functions should be defined for every class (`print, operator<<`) using the same name and with the same general meaning (output the values of all data members in an appropriate format). Taking these two facts together, we see that function-naming conflicts will almost always occur between a derived class and its base class.

   An `extension` is a redefinition of the function that calls the inherited version and also does additional work. Almost every class needs to have a redefinition of the print() function. Normally, this redefinition will call the inherited function using the scope-resolution operator, like this: `return Pubs::print( out );`

   An `override` is a redefinition that fundamentally changes or adds to the meaning of the inherited function or blocks access to it completely, preventing any further derived classes from using it. In each of the two derived classes above, a new method for `print` is defined that overrides the inherited method. As is typical, the function in the derived class prints some data itself, then calls the inherited function to print the rest of the data.

   In a derivation hierarchy, a middle-level class sometimes overrides an inherited function in this way. The effect is to remove that function from the set of functions available to classes further down the inheritance hierarchy. This is a powerful tool, but rarely used.

12.3 Class Derivation Demo

The main program.   We instantiate some Pubs, Printeds and Books and use them to illustrate the syntax and semantics of derivation and inheritance.

```
 1   //------------------------------------------------------------------------------
 2   // Class Derivation and Static Class Members                   file: main.cpp
 3   // A. Fischer March 2, 2009
 4   //------------------------------------------------------------------------------
 5   #include "Pubs.h"
 6   #include "Printed.h"
 7   #include "Book.h"
 8
 9   int Pubs::pubCount = 0; // Number of Pubs objects that now exist.
10   int Printed::prinCount = 0;      // Number of Printed objects that now exist.
11   int Book::bookCount = 0;         // Number of Book objects that now exist.
12
13   //--------------------------------------------------------------------
14   int main( void ) {
15       Pubs A("A-Wesley");
16       Printed B("Trade Books", "Pearson");
17       Book D( "Anatomy", "Out-of-print", "P-Hall" );
18       Book G( "Applied C", "Textbook", "McGraw-Hill" );
19       cout <<"\nThe population is:\n"<< A << B <<  D << G;
20       bye();
21       return 0;
22   }
23   /*  ---------------------------------------------------------------------------
24    *  Pubs: the base class.                                       file: Pubs.h
25    *  Created by Alice Fischer on 3/2/09.
26    */
27   #pragma once;
28   #include "tools.hpp"
29
30   class Pubs {
31       private:    static int pubCount;    // Number of Pubs that exist.
32       protected:  const int serial;       // Serial number of this instance.
33       public:     const char* name;       // Name given in the declaration.
34
35       //----------------------------------------------------------------------------
36       Pubs( char* g ): serial(++pubCount), name(g) {
37           cerr <<"Creating " <<name <<endl;
38       }
39       ~Pubs(){
40           cerr <<"deleting "<<name
41           <<", leaving " <<--pubCount <<" Pubs\n";
42       }
43       ostream& print( ostream& out) const {   // Name, rank, serial number.
44           out <<name <<" #" <<serial <<" out of " <<pubCount <<'\n';
45           return out;
46       }
47   };
48   inline ostream& operator<< (ostream& out, const Pubs& x){ return x.print(out); }
```

12.3.1 Inherited Data Members

Each derivation step adds members to the ones present in the base class. We say that the base-class members are *inherited* by the derived class. An object of the derived class starts with the data members of the base class, followed by the new members declared within the derived class. You could say that the derived object is an extension of the base object. (To do the same derivation in Java, you would write `class Printed extends Pubs`.) Some classes have static data members that are allocated in a non-contiguous part of memory. These members are also inherited.

For example, the first part of a Printed object is a Pubs object. In addition, a Printed has three more data members (name, serial, and prinCount) making a total of six data members in a printed object. Similarly, as shown in the diagram below, the first part of a Book object is a Printed object, to which Book adds another static member (bookCount) and two more ordinary members (name, serial), for a total of nine data members. A book *has* all these members even though some of them are allocated remotely and others are private and cannot be accessed from methods in the Book class.
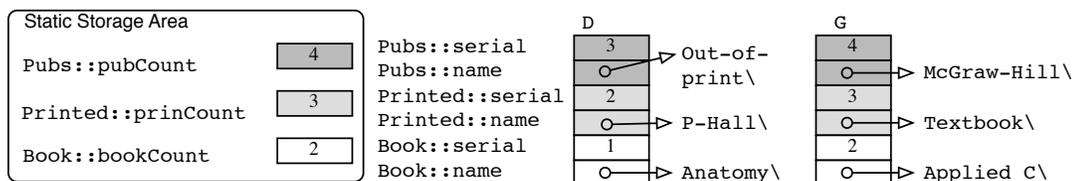

Figure 12.2: A derived object contains an object of its base class.

The data diagram in Figure 12.2 shows the last two objects allocated by `main()`, two Books named `D` and `G`. Each has nine data members: three shared members in the static storage area and six instance members. Medium gray denotes the members that were inherited from the Pubs class; those that came from the Printed class are light gray. Null characters are denoted by a backslash. This diagram is complicated by that fact that all three classes have static members that are stored in another area of storage. The three static members are shared by all `Book` objects in the program.

```
49    /*  -------------------------------------------------------------------------------
50     *  Illustrates private derivation and static class members      file: Printed.h
51     *  Created by Alice Fischer on 3/2/09.
52     */
53    #pragma once;
54    #include "Pubs.h"
55    //---------------------------------------------------------------------------------
56    class Printed : Pubs {
57        private:    static int prinCount;   // Number of Printed objects that exist.
58        protected:  const int serial;       // Serial number of this instance.
59        public:     const char* name;       // Name given in the declaration.
60
61        Printed( char* np, char* pub ): Pubs(pub), serial(++prinCount), name(np) {
62            cerr <<"Creating " <<name <<" based on #" <<Pubs::serial
63            <<" Pubs named " <<Pubs::name <<endl;
64        }
65        ~Printed(){ cerr <<"deleting "<<name <<", leaving " <<--prinCount <<" Printeds\n";}
66
67        ostream& print( ostream& out) const {
68            out <<name <<": " <<" #" <<serial  <<"\n\twhose base object is ";
69            return Pubs::print( out );
70        }
71    };
72    inline ostream& operator<< (ostream& out, const Printed& x){ return x.print(out);}


73    /*  ---------------------------------------------------------------------------
74     *  Public Derivation and static const class member.                file: Book.h
75     *  Created by Alice Fischer on 3/2/09.
76     */
77    #pragma once;
78    #include "Printed.h"
79    //-----------------------------------------------------------------------------
80    class Book : public Printed {
81        private:    static int bookCount;   // Number of Book objects that exist.
82        protected:  const int serial;       // Serial number of this instance.
83        public:     const char* name;       // Name given in the declaration.
84
85        Book(char* b,char* np,char* pub): Printed(np,pub), serial(++bookCount), name(b){
86            cerr <<"Creating " <<name
87            <<" based on #" <<Printed::serial <<" Printed named " <<Printed::name
```

```
88              <<endl;
89          }
90          ~Book(){cerr <<"deleting "<<name <<", leaving " <<--bookCount <<" Book\n";}
91
92          ostream& print( ostream& out) const {
93              out <<name <<" #" <<serial
94                  <<"\n\tis " <<Printed::name <<endl
95                  <<"\tmy Pubs' name is hidden from me." <<endl;
96
97              // out << Pubs::name;        // error: Pubs::name is inaccessible.
98              // Pubs pp = (Pubs)(*this); // error: Pubs is an inaccessible base of Book.
99              // Printed p = (Printed)(*this);   // This line compiles, triggers copying.
100             // out << p.Pubs::name;      // error: Pubs is an inaccessible base of Printed.
101             return out;
102         }
103     };
104     inline ostream& operator<< (ostream& out, const Book& x){ return x.print(out); }
```

Ctors Required.   Every C++ object is built by constructing the base portion, then constructing the composed parts, then calling the class constructor. This means that various parts must be initialized by ctors – doing it in the constructor is too late. In this demo, the constructors for Printed and Book illustrate this principle.

- The constructor for Printed starts on line 61 and has three ctors, all required. The base class, Pubs, does not have a default constructor, so we must pass parameters to it. This is done by a ctor giving the name of the base class, with parentheses enclosing the appropriate arguments: `Pubs(pub)`.

- The serial number is a const int, so it must be initialized in a ctor, not assigned later. This ctor must follow the ctor for the base class, since the base class members form the first part of the object and the ctors must be in order. We see code that increments the shared counter and initializes the serial number: `serial(++prinCount)`.

- The third ctor,   `name(np)` initializes the name field to the string literal written in the program. This does not work for strings read as input, since it does not allocate any new space to store the characters of the string.

- The constructor for Printed starts on line 85, It also has three ctors for its three parts. Note that the ctor for the base class (Printed) has two arguments in parentheses because the Printed constructor needs two parameters.

Access to inherited members.   Various methods in the Printed and Book classes access inherited members.

- The Printed constructor prints trace comments that include two data members from the base class: `Pubs::serial` and `Pubs::name`. The scope-resolution operator must be used to print them because the Printed class has members with the same names.

- These two inherited members are visible in the Printed class because `Pubs::serial` is protected (not private) and `Pubs::name` is public. The derivation method was private derivation, but that does not restrict visibility at this level.

- The Printed::print() method also uses the scope-resolution operator to call the inherited print() method. Without the `Pubs::`, this would be a recursive call (a bug).

- The constructor and print() method in Book do similar things using `Printed::`.

- Since private derivation was used to create Printed, all the members of the Pubs class become private in Printed. Therefore, Book cannot see any of them, even those that were originally public. The evidence for this is on lines 97..100. The compiler will not compile anything in the Book class that tries to breach the privacy of the Pubs class.

The output.   The middle block of output is the "normal" program output. The first block of lines was printed by the constructors, the last block by the destructors. This makes clear the order in which the pieces of these objects are created and deleted. Note that as objects are created, the serial numbers increase in each class that is part of the created object.

```
105    Creating A-Wesley
106    Creating Pearson
107    Creating Trade Books based on #2 Pubs named Pearson
108    Creating P-Hall
109    Creating Out-of-print based on #3 Pubs named P-Hall
110    Creating Anatomy based on #2 Printed named Out-of-print
111    Creating McGraw-Hill
112    Creating Textbook based on #4 Pubs named McGraw-Hill
113    Creating Applied C based on #3 Printed named Textbook
114
115    The population is:
116    A-Wesley #1 out of 4
117    Trade Books:  #1
118         whose base object is Pearson #2 out of 4
119    Anatomy #1
120         is Out-of-print
121         my Pubs' name is hidden from me.
122    Applied C #2
123         is Textbook
124         my Pubs' name is hidden from me.
125
126
127    Normal termination.
128    deleting Applied C, leaving 1 Book
129    deleting Textbook, leaving 2 Printeds
130    deleting McGraw-Hill, leaving 3 Pubs
131    deleting Anatomy, leaving 0 Book
132    deleting Out-of-print, leaving 1 Printeds
133    deleting P-Hall, leaving 2 Pubs
134    deleting Trade Books, leaving 0 Printeds
135    deleting Pearson, leaving 1 Pubs
136    deleting A-Wesley, leaving 0 Pubs
```