

**Chapter
2**

INHERITANCE

*Get on the
Fast Track!*



TM

**SYS-ED/
COMPUTER
EDUCATION
TECHNIQUES, INC.**

Objectives

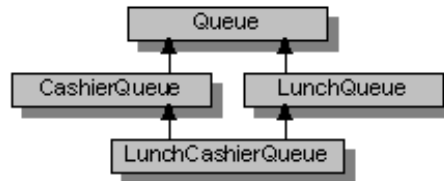
You will learn:

- C Virtual base classes.
- C Single inheritance.
- C Multiple inheritance.
- C Multiple base classes.

1 Virtual Base Classes

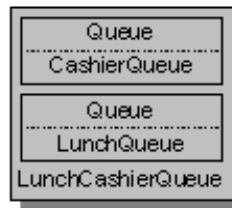
Since a class can be an indirect base class to a derived class more than once, C++ provides a way to optimize the way such base classes work.

Consider the class hierarchy in the following figure, which illustrates a simulated lunch line.



In this diagram, `Queue` is the base class for both `CashierQueue` and `LunchQueue`. However, when both classes are combined to form `LunchCashierQueue`, the following problem arises: the new class contains two subobjects of type `Queue`, one from `CashierQueue` and the other from `LunchQueue`.

This figure illustrates the conceptual memory layout; the actual memory layout might be optimized.



There are two `Queue` subobjects in the `LunchCashierQueue` object.

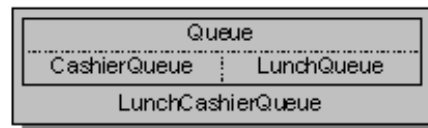
Example:

The following code declares Queue to be a virtual base class:

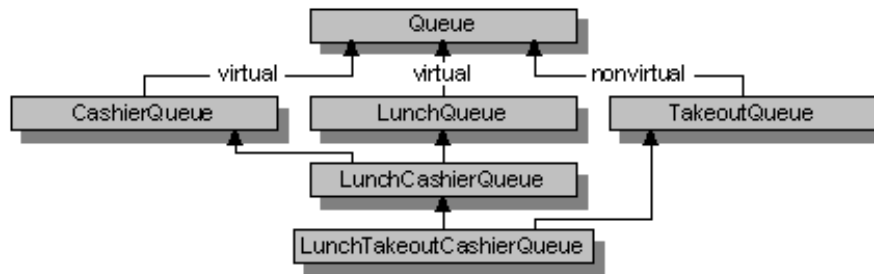
```
1      class Queue
2      {
3          // Member list
4      };
5
6      class CashierQueue : virtual public Queue
7      {
8          // Member list
9      };
10
11     class LunchQueue : virtual public Queue
12     {
13         // Member list
14     };
15
16     class LunchCashierQueue : public LunchQueue, public CashierQueue
17     {
18         // Member list
19     };
```

The virtual keyword ensures that only one copy of the subobject Queue is included.

A class can have both a virtual component and a nonvirtual component of a given type. This happens in the conditions illustrated in the next figure:



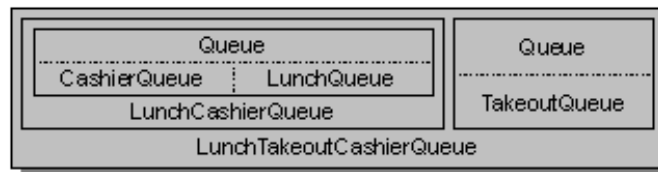
In the figure below, CashierQueue and LunchQueue use Queue as a virtual base class. However, TakeoutQueue specifies Queue as a base class, not a virtual base class.



Therefore, LunchTakeoutCashierQueue has two subobjects of type Queue.

- C One from the inheritance path that includes LunchCashierQueue.
- C One from the path that includes TakeoutQueue.

The two subjects are illustrated in this figure:

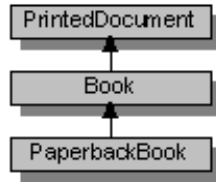


Virtual inheritance provides significant size benefits when compared with nonvirtual inheritance.

However, it can introduce extra processing overhead.

2 Single Inheritance

In "single inheritance," a common form of inheritance, classes have only one base class.



Example:

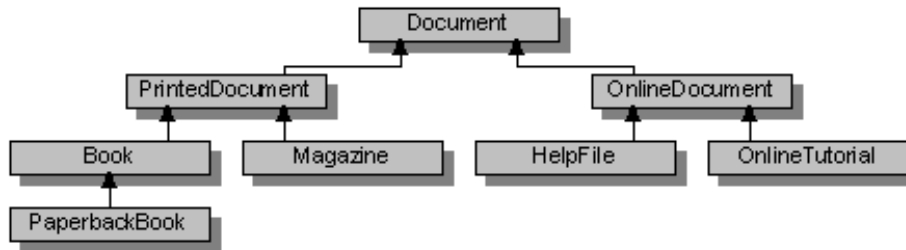
A skeletal declaration of such a class hierarchy is shown in the following code:

```
1      class PrintedDocument
2      {
3          // Member list.
4      };
5
6      // Book is derived from PrintedDocument.
7      class Book : public PrintedDocument
8      {
9          // Member list.
10     };
11
12     // PaperbackBook is derived from Book.
13     class PaperbackBook : public Book
14     {
15         // Member list.
16     };
```

PrintedDocument is considered a "direct base" class to Book; it is an "indirect base" class to PaperbackBook. The difference is that a direct base class appears in the base list of a class declaration and an indirect base does not.

The base class from which each class is derived is declared before the declaration of the derived class. It is not sufficient to provide a forward-referencing declaration for a base class; it must be a complete declaration.

A class can serve as the base class for many specific classes.



In this diagram some of the classes are base classes for more than one derived class. However, the reverse is not true: there is only one direct base class for any given derived class. The graph in the figure depicts a "single inheritance" structure.

Directed acyclic graphs are not unique to single inheritance. They are also used to depict multiple-inheritance graphs.

In inheritance, the derived class contains the members of the base class plus any new members you add. As a result, a derived class can refer to members of the base class; unless those members are redefined in the derived class. The scope-resolution operator (::) can be used to refer to members of direct or indirect base classes when those members have been redefined in the derived class.

Example:

```
1      class Document
2      {
3      public:
4          char *Name;          // Document name.
5          void  PrintNameOf(); // Print name.
6      };
7
8      // Implementation of PrintNameOf function from class Document.
9      void Document::PrintNameOf()
10     {
11         cout << Name << endl;
12     }
13
14     class Book : public Document
15     {
16     public:
17         Book( char *name, long pagecount );
18     private:
19         long  PageCount;
20     };
21
22     // Constructor from class Book.
23     Book::Book( char *name, long pagecount )
24     {
25         Name = new char[ strlen( name ) + 1 ];
26         strcpy( Name, name );
27         PageCount = pagecount;
28     };
```

The constructor for Book, (Book::Book), has access to the data member, Name in a program. An object of type Book can be created and used.

Example:

```
1      // Create a new object of type Book. This invokes the
2      // constructor Book::Book.
3      Book LibraryBook( "Programming Windows, 2nd Ed", 944 );
4
5      ...
6
7      // Use PrintNameOf function inherited from class Document.
8      LibraryBook.PrintNameOf();
```

In the preceding example class-member and inherited data and functions are used identically. If the implementation for class Book calls for a reimplementaion of the PrintNameOf function, the function that belongs to the Document class can be called only by using the scope-resolution (::) operator.

Example:

```
1      class Book : public Document
2      {
3          Book( char *name, long pagecount );
4          void PrintNameOf();
5          long PageCount;
6      };
7      void Book::PrintNameOf()
8      {
9          cout << "Name of book: ";
10         Document::PrintNameOf();
11     }
```

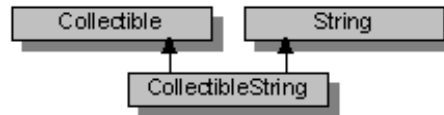
Pointers and references to derived classes can be implicitly converted to pointers and references to their base classes if there is an accessible, unambiguous base class.

The following code demonstrates this concept using pointers; the same principle applies to references:

```
1 #include <iostream.h>
2
3 void main()
4 {
5     Document *DocLib[10]; // Library of ten documents.
6
7     for( int i = 0; i < 10; ++i )
8     {
9         cout << "Type of document: "
10            << "(P)aperback, (M)agazine, (H)elp File, (C)BT"
11            << endl;
12
13         char cDocType;
14         cin >> cDocType;
15
16         switch( tolower( cDocType ) )
17         {
18             case 'p':
19                 DocLib[i] = new PaperbackBook;
20                 break;
21
22             case 'm':
23                 DocLib[i] = new Magazine;
24                 break;
25
26             case 'h':
27                 DocLib[i] = new HelpFile;
28                 break;
29
30             case 'c':
31                 DocLib[i] = new ComputerBasedTraining;
32                 break;
33
34             default:
35                 --i;
36                 break;
37         }
38     }
39
40     for( i = 0; i < 10; ++i )
41         DocLib[i]->PrintNameOf();
42 }
```

3 Multiple Inheritance

Later versions of C++ introduce a "multiple inheritance" model. In a multiple-inheritance graph, the derived classes may have a number of direct base classes.



This diagram shows a class, `CollectibleString`. It is both like a `Collectible` (something that can be contained in a collection), and a `String`. Multiple inheritance is a good solution to the problem where a derived class has attributes of more than one base class. This is because it is easy to form a `CollectibleCustomer`, `CollectibleWindow`, and so on.

If the properties of either class are not required for a particular application, either class can be used alone or in combination with other classes. Therefore, given the hierarchy in the diagram, noncollectible strings and collectibles can be formed which are not strings. This flexibility is not possible using single inheritance.

4 Multiple Base Classes

As with multiple inheritance, a class can be derived from more than one base class. In a multiple-inheritance model; where classes are derived from more than one base class, the base classes are specified using the base-list grammar.

The class declaration for `CollectionOfBook`, derived from `Collection` and `Book`, can be specified:

```
class CollectionOfBook : public Book, public Collection
{
    // New members
};
```

The order in which base classes are specified is not significant except in certain cases where constructors and destructors are invoked.

In these cases, the order in which base classes are specified affects the following:

- C The order in which initialization by constructor takes place.

If the code relies on the `Book` portion of `CollectionOfBook` to be initialized before the `Collection` part, the order of specification is significant. Initialization takes place in the order the classes are specified in the base-list.

- C The order in which destructors are invoked to clean up.

If a particular "part" of the class must be present when the other part is being destroyed, the order is significant. Destructors are called in the reverse order of the classes specified in the base-list.

- C The order of specification of base classes can affect the memory layout of the class.

Do not make any programming decisions based on the order of base members in memory.

When specifying the base-list, the same class name cannot be specified more than once. However, it is possible for a class to be an indirect base to a derived class more than once.