19. Assume a class `Derv` derived from a base class `Base`. Both classes contain a member function `func()` that takes no arguments. Write a statement to go in a member function of `Derv` that calls `func()` in the base class.
20. True or false: It is illegal to make objects of one class members of another class.
21. In the UML, inheritance is called _____.
22. Aggregation is

    a. a stronger form of instantiation.

    b. a stronger form of generalization.

    c. a stronger form of composition.

    d. a "has a" relationship.
23. True or false: the arrow representing generalization points to the more specific class.
24. Composition is a _____ form of _____.

Exercises

Answers to starred exercises can be found in Appendix G.

*1. Imagine a publishing company that markets both book and audiocassette versions of its works. Create a class `publication` that stores the title (a string) and price (type `float`) of a publication. From this class derive two classes: `book`, which adds a page count (type `int`), and `tape`, which adds a playing time in minutes (type `float`). Each of these three classes should have a `getdata()` function to get its data from the user at the keyboard, and a `putdata()` function to display its data.

    Write a `main()` program to test the `book` and `tape` classes by creating instances of them, asking the user to fill in data with `getdata()`, and then displaying the data with `putdata()`.

*2. Recall the STRCONV example from Chapter 8. The `String` class in this example has a flaw: It does not protect itself if its objects are initialized to have too many characters. (The `SZ` constant has the value 80.) For example, the definition

```
String s = "This string will surely exceed the width of the "
        "screen, which is what the SZ constant represents.";
```

    will cause the `str` array in `s` to overflow, with unpredictable consequences, such as crashing the system.

    With `String` as a base class, derive a class `Pstring` (for "protected string") that prevents buffer overflow when too long a string constant is used in a definition. A new constructor in the derived class should copy only `SZ-1` characters into `str` if the string constant is longer, but copy the entire constant if it's shorter. Write a `main()` program to test different lengths of strings.

*3. Start with the `publication`, `book`, and `tape` classes of Exercise 1. Add a base class `sales` that holds an array of three `floats` so that it can record the dollar sales of a particular publication for the last three months. Include a `getdata()` function to get three sales amounts from the user, and a `putdata()` function to display the sales figures. Alter the `book` and `tape` classes so they are derived from both `publication` and `sales`. An object of class `book` or `tape` should input and output sales data along with its other data. Write a `main()` function to create a `book` object and a `tape` object and exercise their input/output capabilities.

4. Assume that the publisher in Exercises 1 and 3 decides to add a third way to distribute books: on computer disk, for those who like to do their reading on their laptop. Add a `disk` class that, like `book` and `tape`, is derived from `publication`. The `disk` class should incorporate the same member functions as the other classes. The data item unique to this class is the disk type: either CD or DVD. You can use an `enum` type to store this item. The user could select the appropriate type by typing `c` or `d`.

5. Derive a class called `employee2` from the `employee` class in the EMPLOY program in this chapter. This new class should add a type `double` data item called `compensation`, and also an `enum` type called `period` to indicate whether the employee is paid hourly, weekly, or monthly. For simplicity you can change the `manager`, `scientist`, and `laborer` classes so they are derived from `employee2` instead of `employee`. However, note that in many circumstances it might be more in the spirit of OOP to create a separate base class called `compensation` and three new classes `manager2`, `scientist2`, and `laborer2`, and use multiple inheritance to derive these three classes from the original `manager`, `scientist`, and `laborer` classes and from `compensation`. This way none of the original classes needs to be modified.

6. Start with the ARROVER3 program in Chapter 8. Keep the `safearay` class the same as in that program, and, using inheritance, derive the capability for the user to specify both the upper and lower bounds of the array in a constructor. This is similar to Exercise 9 in Chapter 8, except that inheritance is used to derive a new class (you can call it `safehilo`) instead of modifying the original class.

7. Start with the COUNTEN2 program in this chapter. It can increment or decrement a counter, but only using prefix notation. Using inheritance, add the ability to use postfix notation for both incrementing and decrementing. (See Chapter 8 for a description of postfix notation.)

8. Operators in some computer languages, such as Visual Basic, allow you to select parts of an existing string and assign them to other strings. (The Standard C++ `string` class offers a different approach.) Using inheritance, add this capability to the `Pstring` class of Exercise 2. In the derived class, `Pstring2`, incorporate three new functions: `left()`, `mid()`, and `right()`.

```
s2.left(s1, n)    // s2 is assigned the leftmost n characters
                  //    from s1
s2.mid(s1, s, n)  // s2 is assigned the middle n characters
                  //    from s1, starting at character number s
                  //    (leftmost character is 0)
s2.right(s1, n)   // s2 is assigned the rightmost n characters
                  //    from s1
```

You can use `for` loops to copy the appropriate parts of `s1`, character by character, to a temporary `Pstring2` object, which is then returned. For extra credit, have these functions return by reference, so they can be used on the left side of the equal sign to change parts of an existing string.

9. Start with the `publication`, `book`, and `tape` classes of Exercise 1. Suppose you want to add the date of publication for both books and tapes. From the `publication` class, derive a new class called `publication2` that includes this member data. Then change `book` and `tape` so they are derived from `publication2` instead of `publication`. Make all the necessary changes in member functions so the user can input and output dates along with the other data. For the dates, you can use the `date` class from Exercise 5 in Chapter 6, which stores a date as three `int`s, for month, day, and year.
10. There is only one kind of manager in the EMPMULT program in this chapter. Any serious company has executives as well as managers. From the `manager` class derive a class called `executive`. (We'll assume an executive is a high-end kind of manager.) The additional data in the `executive` class will be the size of the employee's yearly bonus and the number of shares of company stock held in his or her stock-option plan. Add the appropriate member functions so these data items can be input and displayed along with the other `manager` data.
11. Various situations require that pairs of numbers be treated as a unit. For example, each screen coordinate has an x (horizontal) component and a y (vertical) component. Represent such a pair of numbers as a structure called `pair` that comprises two `int` member variables.

    Now, assume you want to be able to store `pair` variables on a stack. That is, you want to be able to place a pair (which contains two integers) onto a stack using a single call to a `push()` function with a structure of type `pair` as an argument, and retrieve a pair using a single call to a `pop()` function, which will return a structure of type `pair`. Start with the `Stack2` class in the STAKEN program in this chapter, and from it derive a new class called `pairStack`. This new class need contain only two members: the overloaded `push()` and `pop()` functions. The `pairStack::push()` function will need to make two calls to `Stack2::push()` to store the two integers in its pair, and the `pairStack::pop()` function will need to make two calls to `Stack2::pop()` (although not necessarily in the same order).

12. Amazing as it may seem, the old British pounds-shillings-pence money notation (£9.19.11—see Exercise 10 in Chapter 4, "Structures") isn't the whole story. A penny was further divided into halfpennies and farthings, with a farthing being worth 1/4 of a penny. There was a halfpenny coin, a farthing coin, and a halffarthing coin. Fortunately all this can be expressed numerically in eighths of a penny:

    1/8 penny is a halffarthing

    1/4 penny is a farthing

    3/8 penny is a farthing and a half

    1/2 penny is a halfpenny (pronounced ha'penny)

    5/8 penny is a halfpenny plus a halffarthing

    3/4 penny is a halfpenny plus a farthing

    7/8 penny is a halfpenny plus a farthing and a half

    Let's assume we want to add to the `sterling` class the ability to handle such fractional pennies. The I/O format can be something like `£1.1.1-1/4` or `£9.19.11-7/8`, where the hyphen separates the fraction from the pennies.

    Derive a new class called `sterfrac` from `sterling`. It should be able to perform the four arithmetic operations on sterling quantities that include eighths of a penny. Its only member data is an `int` indicating the number of eighths; you can call it `eighths`. You'll need to overload many of the functions in `sterling` to handle the eighths. The user should be able to type any fraction in lowest terms, and the display should also show fractions in lowest terms. It's not necessary to use the full-scale `fraction` class (see Exercise 11 in Chapter 6), but you could try that for extra credit.