22. In the UML, member data items are called _____ and member functions are called _____.
23. True or false: rectangles that symbolize classes have rounded corners.
24. Navigability from class A to class B means that

    a. an object of class A can call an operation in an object of class B.

    b. there is a relationship between class A and class B.

    c. objects can go from class A to class B.

    d. messages from class B are received by class A.

Exercises

Answers to starred exercises can be found in Appendix G.

*1. To the `Distance` class in the ENGLPLUS program in this chapter, add an overloaded `-` operator that subtracts two distances. It should allow statements like `dist3=dist1-dist2;`. Assume that the operator will never be used to subtract a larger number from a smaller one (that is, negative distances are not allowed).

*2. Write a program that substitutes an overloaded `+=` operator for the overloaded `+` operator in the STRPLUS program in this chapter. This operator should allow statements like

```
s1 += s2;
```

where `s2` is added (concatenated) to `s1` and the result is left in `s1`. The operator should also permit the results of the operation to be used in other calculations, as in

```
s3 = s1 += s2;
```

*3. Modify the `time` class from Exercise 3 in Chapter 6 so that instead of a function `add_time()` it uses the overloaded `+` operator to add two times. Write a program to test this class.

*4. Create a class `Int` based on Exercise 1 in Chapter 6. Overload four integer arithmetic operators (`+`, `-`, `*`, and `/`) so that they operate on objects of type `Int`. If the result of any such arithmetic operation exceeds the normal range of `int`s (in a 32-bit environment)—from 2,147,483,648 to –2,147,483,647—have the operator print a warning and terminate the program. Such a data type might be useful where mistakes caused by arithmetic overflow are unacceptable. Hint: To facilitate checking for overflow, perform the calculations using type `long double`. Write a program to test this class.

5. Augment the `time` class referred to in Exercise 3 to include overloaded increment (`++`) and decrement (`--`) operators that operate in both prefix and postfix notation and return values. Add statements to `main()` to test these operators.

6. Add to the `time` class of Exercise 5 the ability to subtract two time values using the overloaded (`-`) operator, and to multiply a time value by a number of type `float`, using the overloaded (`*`) operator.
7. Modify the `fraction` class in the four-function fraction calculator from Exercise 11 in Chapter 6 so that it uses overloaded operators for addition, subtraction, multiplication, and division. (Remember the rules for fraction arithmetic in Exercise 12 in Chapter 3, "Loops and Decisions.") Also overload the == and `!=` comparison operators, and use them to exit from the loop if the user enters 0/1, 0/1 for the values of the two input fractions. You may want to modify the `lowterms()` function so that it returns the value of its argument reduced to lowest terms. This makes it more useful in the arithmetic functions, where it can be applied just before the answer is returned.
8. Modify the `bMoney` class from Exercise 12 in Chapter 7, "Arrays and Strings," to include the following arithmetic operations, performed with overloaded operators:

```
bMoney = bMoney + bMoney
bMoney = bMoney - bMoney
bMoney = bMoney * long double  (price per widget times number of widgets)
long double = bMoney / bMoney  (total price divided by price per widget)
bMoney = bMoney / long double  (total price divided by number of widgets)
```

   Notice that the `/` operator is overloaded twice. The compiler can distinguish between the two usages because the arguments are different. Remember that it's easy to perform arithmetic operations on `bMoney` objects by performing the same operation on their `long double` data.

   Make sure the `main()` program asks the user to enter two money strings and a floating-point number. It should then carry out all five operations and display the results. This should happen in a loop, so the user can enter more numbers if desired.

   Some money operations don't make sense: `bMoney * bMoney` doesn't represent anything real, since there is no such thing as square money; and you can't add `bMoney` to `long double` (what's dollars plus widgets?). To make it impossible to compile such illegal operations, don't include conversion operators for `bMoney` to `long double` or `long double` to `bMoney`. If you do, and you write an expression like

```
bmon2 = bmon1 + widgets;  // doesn't make sense
```

   then the compiler will automatically convert widgets to `bMoney` and carry out the addition. Without them, the compiler will flag such conversions as errors, making it easier to catch conceptual mistakes. Also, make any conversion constructors explicit.

   There are some other plausible money operations that we don't yet know how to perform with overloaded operators, since they require an object on the right side of the operator but not the left:

```
long double * bMoney  // can't do this yet: bMoney only on right
long double / bMoney  // can't do this yet: bMoney only on right
```

   We'll learn how to handle this situation when we discuss friend functions in Chapter 11.

9. Augment the `safearay` class in the ARROVER3 program in this chapter so that the user can specify both the upper and lower bound of the array (indexes running from 100 to 200, for example). Have the overloaded subscript operator check the index each time the array is accessed to ensure that it is not out of bounds. You'll need to add a two-argument constructor that specifies the upper and lower bounds. Since we have not yet learned how to allocate memory dynamically, the member data will still be an array that starts at 0 and runs up to 99, but perhaps you can map the indexes for the `safearay` into different indexes in the real `int` array. For example, if the client selects a range from 100 to 175, you could map this into the range from `arr[0]` to `arr[75]`.
10. For math buffs only: Create a class `Polar` that represents the points on the plain as polar coordinates (radius and angle). Create an overloaded `+operator` for addition of two `Polar` quantities. "Adding" two points on the plain can be accomplished by adding their X coordinates and then adding their Y coordinates. This gives the X and Y coordinates of the "answer." Thus you'll need to convert two sets of polar coordinates to rectangular coordinates, add them, then convert the resulting rectangular representation back to polar.
11. Remember the `sterling` structure? We saw it in Exercise 10 in Chapter 2, "C++ Programming Basics," and in Exercise 11 in Chapter 5, among other places. Turn it into a class, with pounds (type `long`), shillings (type `int`), and pence (type `int`) data items. Create the following member functions:
    - no-argument constructor
    - one-argument constructor, taking type `double` (for converting from decimal pounds)
    - three-argument constructor, taking pounds, shillings, and pence
    - `getSterling()` to get an amount in pounds, shillings, and pence from the user, format £9.19.11
    - `putSterling()` to display an amount in pounds, shillings, and pence, format £9.19.11
    - addition (`sterling + sterling`) using overloaded `+` operator
    - subtraction (`sterling - sterling`) using overloaded `-` operator
    - multiplication (`sterling * double`) using overloaded `*` operator
    - division (`sterling / sterling`) using overloaded `/` operator
    - division (`sterling / double`) using overloaded `/` operator
    - operator `double` (to convert to `double`)

    To perform arithmetic, you could (for example) add each object's data separately: Add the pence, carry, add the shillings, carry, and so on. However, it's easier to use the conversion operator to convert both `sterling` objects to type `double`, perform the arithmetic on the doubles, and convert back to `sterling`. Thus the overloaded `+` operator looks like this:

```
sterling sterling::operator + (sterling s2)
   {
   return sterling( double(sterling(pounds, shillings, pence))
                   + double(s2) );
   }
```

This creates two temporary `double` variables, one derived from the object of which the function is a member, and one derived from the argument `s2`. These `double` variables are then added, and the result is converted back to `sterling` and returned.

Notice that we use a different philosophy with the `sterling` class than with the `bMoney` class. With sterling we use conversion operators, thus giving up the ability to catch illegal math operations but gaining simplicity in writing the overloaded math operators.

12. Write a program that incorporates both the `bMoney` class from Exercise 8 and the `sterling` class from Exercise 11. Write conversion operators to convert between `bMoney` and `sterling`, assuming that one pound (£1.0.0) equals fifty dollars ($50.00). This was the approximate exchange rate in the 19th century when the British Empire was at its height and the pounds-shillings-pence format was in use. Write a `main()` program that allows the user to enter an amount in either currency and then converts it to the other currency and displays the result. Minimize any modifications to the existing `bMoney` and `sterling` classes.