20. A string in C++ is an _____ of type _____.
21. Write a statement that defines a string variable called `city` that can hold a string of up to 20 characters (this is slightly tricky).
22. Write a statement that defines a string constant, called `dextrose`, that has the value "C6H12O6-H2O".
23. True or false: The extraction operator (>>) stops reading a string when it encounters a space.
24. You can read input that consists of multiple lines of text using

    a. the normal `cout <<` combination.

    b. the `cin.get()` function with one argument.

    c. the `cin.get()` function with two arguments.

    d. the `cin.get()` function with three arguments.
25. Write a statement that uses a string library function to copy the string `name` to the string `blank`.
26. Write the declaration for a class called `dog` that contains two data members: a string called `breed` and an `int` called `age`. (Don't include any member functions.)
27. True or false: You should prefer C-strings to the Standard C++ `string` class in new programs.
28. Objects of the `string` class

    a. are zero-terminated.

    b. can be copied with the assignment operator.

    c. do not require memory management.

    d. have no member functions.
29. Write a statement that finds where the string `"cat"` occurs in the string `s1`.
30. Write a statement that inserts the string `"cat"` into string `s1` at position 12.

Exercises

Answers to the starred exercises can be found in Appendix G.

*1. Write a function called `reversit()` that reverses a C-string (an array of `char`). Use a `for` loop that swaps the first and last characters, then the second and next-to-last characters, and so on. The string should be passed to `reversit()` as an argument.

    Write a program to exercise `reversit()`. The program should get a string from the user, call `reversit()`, and print out the result. Use an input method that allows embedded blanks. Test the program with Napoleon's famous phrase, "Able was I ere I saw Elba."

*2. Create a class called `employee` that contains a name (an object of class `string`) and an employee number (type `long`). Include a member function called `getdata()` to get data from the user for insertion into the object, and another function called `putdata()` to display the data. Assume the name has no embedded blanks.

   Write a `main()` program to exercise this class. It should create an array of type `employee`, and then invite the user to input data for up to 100 employees. Finally, it should print out the data for all the employees.

*3. Write a program that calculates the average of up to 100 English distances input by the user. Create an array of objects of the `Distance` class, as in the ENGLARAY example in this chapter. To calculate the average, you can borrow the `add_dist()` member function from the ENGLCON example in Chapter 6. You'll also need a member function that divides a `Distance` value by an integer. Here's one possibility:

```
void Distance::div_dist(Distance d2, int divisor)
   {
   float fltfeet = d2.feet + d2.inches/12.0;
   fltfeet /= divisor;
   feet = int(fltfeet);
   inches = (fltfeet-feet) * 12.0;
   }
```

4. Start with a program that allows the user to input a number of integers, and then stores them in an `int` array. Write a function called `maxint()` that goes through the array, element by element, looking for the largest one. The function should take as arguments the address of the array and the number of elements in it, and return the index number of the largest element. The program should call this function and then display the largest element and its index number. (See the SALES program in this chapter.)
5. Start with the `fraction` class from Exercises 11 and 12 in Chapter 6. Write a `main()` program that obtains an arbitrary number of fractions from the user, stores them in an array of type `fraction`, averages them, and displays the result.
6. In the game of contract bridge, each of four players is dealt 13 cards, thus exhausting the entire deck. Modify the CARDARAY program in this chapter so that, after shuffling the deck, it deals four hands of 13 cards each. Each of the four players' hands should then be displayed.
7. One of the weaknesses of C++ for writing business programs is that it does not contain a built-in type for monetary values such as $173,698,001.32. Such a money type should be able to store a number with a fixed decimal point and about 17 digits of precision, which is enough to handle the national debt in dollars and cents. Fortunately, the built-in C++ type `long double` has 19 digits of precision, so we can use it as the basis of a money class, even though it uses a floating decimal. However, we'll need to add the capability to input and output money amounts preceded by a dollar sign and divided by commas into

groups of three digits; this makes it much easier to read large numbers. As a first step toward developing such a class, write a function called `mstold()` that takes a *money string*, a string representing a money amount like

```
"$1,234,567,890,123.99"
```

as an argument, and returns the equivalent `long double`.

You'll need to treat the money string as an array of characters, and go through it character by character, copying only digits (1–9) and the decimal point into another string. Ignore everything else, including the dollar sign and the commas. You can then use the `_atold()` library function (note the initial underscore—header file STDLIB.H or MATH.H) to convert the resulting pure string to a `long double`. Assume that money values will never be negative. Write a `main()` program to test `mstold()` by repeatedly obtaining a money string from the user and displaying the corresponding `long double`.

8. Another weakness of C++ is that it does not automatically check array indexes to see whether they are in bounds. (This makes array operations faster but less safe.) We can use a class to create a safe array that checks the index of all array accesses.

   Write a class called `safearay` that uses an `int` array of fixed size (call it `LIMIT`) as its only data member. There will be two member functions. The first, `putel()`, takes an index number and an `int` value as arguments and inserts the `int` value into the array at the index. The second, `getel()`, takes an index number as an argument and returns the `int` value of the element with that index.

   ```
   safearay sa1;           // define a safearay object
   int temp = 12345;       // define an int value
   sa1.putel(7, temp);     // insert value of temp into array at index 7
   temp = sa1.getel(7);    // obtain value from array at index 7
   ```

   Both functions should check the index argument to make sure it is not less than 0 or greater than `LIMIT-1`. You can use this array without fear of writing over other parts of memory.

   Using functions to access array elements doesn't look as eloquent as using the `[]` operator. In Chapter 8 we'll see how to overload this operator to make our `safearay` class work more like built-in arrays.

9. A queue is a data storage device much like a stack. The difference is that in a stack the last data item stored is the first one retrieved, while in a queue the first data item stored is the first one retrieved. That is, a stack uses a last-in-first-out (LIFO) approach, while a queue uses first-in-first-out (FIFO). A queue is like a line of customers in a bank: The first one to join the queue is the first one served.

   Rewrite the STAKARAY program from this chapter to incorporate a class called `queue` instead of a class called `stack`. Besides a constructor, it should have two functions: one called `put()` to put a data item on the queue, and one called `get()` to get data from the queue. These are equivalent to `push()` and `pop()` in the `stack` class.

Both a queue and a stack use an array to hold the data. However, instead of a single `int` variable called `top`, as the stack has, you'll need two variables for a queue: one called `head` to point to the head of the queue, and one called `tail` to point to the tail. Items are placed on the queue at the tail (like the last customer getting in line at the bank) and removed from the queue at the head. The tail will follow the head along the array as items are added and removed from the queue. This results in an added complexity: When either the tail or the head gets to the end of the array, it must wrap around to the beginning. Thus you'll need a statement like

```
if(tail == MAX-1)
   tail = -1;
```

to wrap the tail, and a similar one for the head. The array used in the queue is sometimes called a circular buffer, because the head and tail circle around it, with the data between them.

10. A matrix is a two-dimensional array. Create a class `matrix` that provides the same safety feature as the array class in Exercise 7; that is, it checks to be sure no array index is out of bounds. Make the member data in the `matrix` class a 10-by-10 array. A constructor should allow the programmer to specify the actual dimensions of the matrix (provided they're less than 10 by 10). The member functions that access data in the matrix will now need two index numbers: one for each dimension of the array. Here's what a fragment of a `main()` program that operates on such a class might look like:

```
matrix m1(3, 4);           // define a matrix object
int temp = 12345;          // define an int value
m1.putel(7, 4, temp);      // insert value of temp into matrix at 7,4
temp = m1.getel(7, 4);     // obtain value from matrix at 7,4
```

11. Refer back to the discussion of money strings in Exercise 6. Write a function called `ldtoms()` to convert a number represented as type `long double` to the same value represented as a money string. First you should check that the value of the original long double is not too large. We suggest that you don't try to convert any number greater than 9,999,999,999,999,990.00. Then convert the `long double` to a pure string (no dollar sign or commas) stored in memory, using an `ostrstream` object, as discussed earlier in this chapter. The resulting formatted string can go in a buffer called `ustring`.

    You'll then need to start another string with a dollar sign; copy one digit from `ustring` at a time, starting from the left, and inserting a comma into the new string every three digits. Also, you'll need to suppress leading zeros. You want to display $3,124.95, for example, not $0,000,000,000,003,124.95. Don't forget to terminate the string with a `'\0'` character.

    Write a `main()` program to exercise this function by having the user repeatedly input numbers in type `long double` format, and printing out the result as a money string.

12. Create a class called `bMoney`. It should store money amounts as `long doubles`. Use the function `mstold()` to convert a money string entered as input into a `long double`, and the function `ldtoms()` to convert the `long double` to a money string for display. (See Exercises 6 and 10.) You can call the input and output member functions `getmoney()` and `putmoney()`. Write another member function that adds two `bMoney` amounts; you can call it `madd()`. Adding `bMoney` objects is easy: Just add the `long double` member data amounts in two `bMoney` objects. Write a `main()` program that repeatedly asks the user to enter two money strings, and then displays the sum as a money string. Here's how the class `specifier` might look:

```
class bMoney
    {
    private:
        long double money;
    public:
        bMoney();
        bMoney(char s[]);
        void madd(bMoney m1, bMoney m2);
        void getmoney();
        void putmoney();
    };
```