33. True or false: a transition between states exists for the duration of the program.
34. A guard in a state diagram is

    a. a constraint on when a transition can occur.

    b. a name for certain kinds of transitions.

    c. a name for certain kinds of states.

    d. a restriction on the creation of certain states.

Exercises

Answers to starred exercises can be found in Appendix G.

*1. Write a program that reads a group of numbers from the user and places them in an array of type `float`. Once the numbers are stored in the array, the program should average them and print the result. Use pointer notation wherever possible.

*2. Start with the `String` class from the NEWSTR example in this chapter. Add a member function called `upit()` that converts the string to all uppercase. You can use the `toupper()` library function, which takes a single character as an argument and returns a character that has been converted (if necessary) to uppercase. This function uses the CCTYPE header file. Write some code in `main()` to test `upit()`.

*3. Start with an array of pointers to strings representing the days of the week, as found in the PTRTOSTR program in this chapter. Provide functions to sort the strings into alphabetical order, using variations of the `bsort()` and `order()` functions from the PTRSORT program in this chapter. Sort the pointers to the strings, not the actual strings.

*4. Add a destructor to the LINKLIST program. It should delete all the links when a `linklist` object is destroyed. It can do this by following along the chain, deleting each link as it goes. You can test the destructor by having it display a message each time it deletes a link; it should delete the same number of links that were added to the list. (A destructor is called automatically by the system for any existing objects when the program exits.)

5. Suppose you have a `main()` with three local arrays, all the same size and type (say `float`). The first two are already initialized to values. Write a function called `addarrays()` that accepts the addresses of the three arrays as arguments; adds the contents of the first two arrays together, element by element; and places the results in the third array before returning. A fourth argument to this function can carry the size of the arrays. Use pointer notation throughout; the only place you need brackets is in defining the arrays.

6. Make your own version of the library function `strcmp(s1, s2)`, which compares two strings and returns –1 if `s1` comes first alphabetically, 0 if `s1` and `s2` are the same, and 1 if `s2` comes first alphabetically. Call your function `compstr()`. It should take two `char*` strings as arguments, compare them character by character, and return an `int`. Write a `main()` program to test the function with different combinations of strings. Use pointer notation throughout.

7. Modify the `person` class in the PERSORT program in this chapter so that it includes not only a name, but also a `salary` item of type `float` representing the person's salary. You'll need to change the `setName()` and `printName()` member functions to `setData()` and `printData()`, and include in them the ability to set and display the salary as well as the name. You'll also need a `getSalary()` function. Using pointer notation, write a `salsort()` function that sorts the pointers in the `persPtr` array by salary rather than by name. Try doing all the sorting in `salsort()`, rather than calling another function as PERSORT does. If you do this, don't forget that `->` takes precedence over `*`, so you'll need to say

   ```
   if( (*(pp+j))->getSalary() > (*(pp+k))->getSalary() )
       { /* swap the pointers */ }
   ```

8. Revise the `additem()` member function from the LINKLIST program so that it adds the item at the end of the list, rather than the beginning. This will cause the first item inserted to be the first item displayed, so the output of the program will be

   ```
   25
   36
   49
   64
   ```

   To add the item, you'll need to follow the chain of pointers to the end of the list, then change the last link to point to the new link.

9. Let's say that you need to store 100 integers so that they're easily accessible. However, let's further assume that there's a problem: The memory in your computer is so fragmented that the largest array that you can use holds only 10 integers. (Such problems actually arise, although usually with larger memory objects.) You can solve this problem by defining 10 separate `int` arrays of 10 integers each, and an array of 10 pointers to these arrays. The `int` arrays can have names like `a0`, `a1`, `a2`, and so on. The address of each of these arrays can be stored in the pointer array of type `int*`, which can have a name like `ap` (for array of pointers). You can then access individual integers using expressions like `ap[j][k]`, where `j` steps through the pointers in `ap` and `k` steps through individual integers in each array. This looks as if you're accessing a two-dimensional array, but it's really a group of one-dimensional arrays.

   Fill such a group of arrays with test data (say the numbers 0, 10, 20, and so on up to 990). Then display the data to make sure it's correct.

10. As presented, Exercise 9 is rather inelegant because each of the 10 `int` arrays is declared in a different program statement, using a different name. Each of their addresses must also be obtained using a separate statement. You can simplify things by using `new`, which allows you to allocate the arrays in a loop and assign pointers to them at the same time:

```
for(j=0; j<NUMARRAYS; j++)          // allocate NUMARRAYS arrays
   *(ap+j) = new int[MAXSIZE];      // each MAXSIZE ints long
```

    Rewrite the program in Exercise 9 to use this approach. You can access the elements of the individual arrays using the same expression mentioned in Exercise 9, or you can use pointer notation: `*(*(ap+j)+k)`. The two notations are equivalent.

11. Create a class that allows you to treat the 10 separate arrays in Exercise 10 as a single one-dimensional array, using array notation with a single index. That is, statements in `main()` can access their elements using expressions like `a[j]`, even though the class member functions must access the data using the two-step approach. Overload the subscript operator `[]` (see Chapter 9, "Inheritance") to achieve this result. Fill the arrays with test data and then display it. Although array notation is used in the class interface in `main()` to access "array" elements, you should use only pointer notation for all the operations in the implementation (within the class member functions).

12. Pointers are complicated, so let's see whether we can make their operation more understandable (or possibly more impenetrable) by simulating their operation with a class.

    To clarify the operation of our homemade pointers, we'll model the computer's memory using arrays. This way, since array access is well understood, you can see what's really going on when we access memory with pointers.

    We'd like to use a single array of type `char` to store all types of variables. This is what a computer memory really is: an array of bytes (which are the same size as type `char`), each of which has an address (or, in array-talk, an index). However, C++ won't ordinarily let us store a `float` or an `int` in an array of type `char`. (We could use unions, but that's another story.) So we'll simulate memory by using a separate array for each data type we want to store. In this exercise we'll confine ourselves to one numerical type, `float`, so we'll need an array of this type; call it `fmemory`. However, pointer values (addresses) are also stored in memory, so we'll need another array to store them. Since we're using array indexes to model addresses, and indexes for all but the largest arrays can be stored in type `int`, we'll create an array of this type (call it `pmemory`) to hold these "pointers."

    An index to `fmemory` (call it `fmem_top`) points to the next available place where a `float` value can be stored. There's a similar index to `pmemory` (call it `pmem_top`). Don't worry about running out of "memory." We'll assume these arrays are big enough so that each time we store something we can simply insert it at the next index number in the array. Other than this, we won't worry about memory management.

Create a class called `Float`. We'll use it to model numbers of type `float` that are stored in `fmemory` instead of real memory. The only instance data in `Float` is its own "address"; that is, the index where its `float` value is stored in `fmemory`. Call this instance variable `addr`. Class `Float` also needs two member functions. The first is a one-argument constructor to initialize the `Float` with a `float` value. This constructor stores the `float` value in the element of `fmemory` pointed to by `fmem_top`, and stores the value of `fmem_top` in `addr`. This is similar to how the compiler and linker arrange to store an ordinary variable in real memory. The second member function is the overloaded `&` operator. It simply returns the pointer (really the index, type `int`) value in `addr`.

Create a second class called `ptrFloat`. The instance data in this class holds the address (index) in `pmemory` where some other address (index) is stored. A member function initializes this "pointer" with an `int` index value. The second member function is the overloaded `*` (dereference, or "contents of") operator. Its operation is a tad more complicated. It obtains the address from `pmemory`, where its data, which is also an address, is stored. It then uses this new address as an index into `fmemory` to obtain the float value pointed to by its address data.

```
float& ptrFloat::operator*()
   {
   return fmemory[ pmemory[addr] ];
   }
```

In this way it models the operation of the dereference operator (`*`). Notice that you need to return by reference from this function so that you can use `*` on the left side of the equal sign.

The two classes `Float` and `ptrFloat` are similar, but `Float` stores `float`s in an array representing memory, and `ptrFloat` stores `int`s (representing memory pointers, but really array index values) in a different array that also represents memory.

Here's a typical use of these classes, from a sample `main()`:

```
Float var1 = 1.234;          // define and initialize two Floats
Float var2 = 5.678;

ptrFloat ptr1 = &var1;       // define two pointers-to-Floats,
ptrFloat ptr2 = &var2;       // initialize to addresses of Floats

cout << " *ptr1=" << *ptr1;  // get values of Floats indirectly
cout << " *ptr2=" << *ptr2;  // and display them

*ptr1 = 7.123;               // assign new values to variables
*ptr2 = 8.456;               // pointed to by ptr1 and ptr2
```

```
cout << " *ptr1=" << *ptr1;  // get new values indirectly
cout << " *ptr2=" << *ptr2;  // and display them
```

Notice that, aside from the different names for the variable types, this looks just the same as operations on real variables. Here's the output from the program:

```
*ptr1=1.234
*ptr2=2.678

*ptr1=7.123
*ptr2=8.456
```

This may seem like a roundabout way to implement pointers, but by revealing the inner workings of the pointer and address operator, we have provided a different perspective on their true nature.