Chapter 15 Pointers: Beyond the Basics

15-1. Pointers to Pointers

As you learned on Day 9, a *pointer* is a numeric variable with a value that is the address of another variable. You declare a pointer using the indirection operator (*). For example, the declaration

```
int *ptr;
```

declares a pointer named ptr that can point to a type int variable. You then use the address-of operator (&) to make the pointer point to a specific variable of the corresponding type. Assuming that x has been declared as a type int variable, the statement

```
ptr = &x;
```

assigns the address of x to ptr and makes ptr point to x. Again, using the indirection operator, you can access the pointed-to variable by using its pointer. Both of the following statements assign the value 12 to x:

```
x = 12;
*ptr = 12;
```

Because a pointer is itself a numeric variable, it is stored in your computer's memory at a particular address. Therefore, you can create a pointer to a pointer, a variable whose value is the address of a pointer. Here's how:

```
int x = 12;                    /* x is a type int variable. */
int *ptr = &x;                 /* ptr is a pointer to x. */
int **ptr_to_ptr = &ptr;       /* ptr_to_ptr is a pointer to a */
                               /* pointer to type int. */
```

Note the use of a double indirection operator (**) when declaring a pointer to a pointer. You also use the double indirection operator when accessing the pointed-to variable with a pointer to a pointer. Thus, the statement

```
**ptr_to_ptr = 12;
```

assigns the value 12 to the variable x, and the statement

```
printf("%d", **ptr_to_ptr);
```

displays the value of x on-screen. If you mistakenly use a single indirection operator, you get errors. The statement

```
*ptr_to_ptr = 12;
```

assigns the value 12 to ptr, which results in ptr's pointing to whatever happens to be stored at address 12. This clearly is a mistake.

Declaring and using a pointer to a pointer is called *multiple indirection*. Figure 15.1 shows the relationship between a variable, a pointer, and a pointer to a pointer. There's really no limit to the level of multiple indirection possible--you can have a pointer to a pointer to a pointer *ad infinitum,* but there's rarely any advantage to going beyond two levels; the complexities involved are an invitation to mistakes.

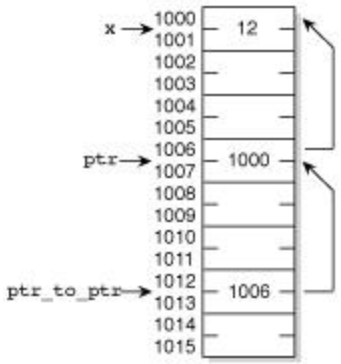

Figure 15-1. A pointer to a pointer.

What can you use pointers to pointers for? The most common use involves arrays of pointers, which are covered later in this chapter. Listing 19.5 on Day 19, "Exploring the C Function Library," presents an example of using multiple indirection.

15-2. Pointers and Multidimensional Arrays

Day 8, "Using Numeric Arrays," covers the special relationship between pointers and arrays. Specifically, the name of an array without its following brackets is a pointer to the first element of the array. As a result, it's easier to use pointer notation when you're accessing certain types of arrays. These earlier examples, however, were limited to single-dimensional arrays. What about multidimensional arrays?

Remember that a multidimensional array is declared with one set of brackets for each dimension. For example, the following statement declares a two-dimensional array that contains eight type int variables:

```
int multi[2][4];
```

You can visualize an array as having a row and column structure--in this case, two rows and four columns. There's another way to visualize a multidimensional array, however, and this way is closer to the manner in which C actually handles arrays. You can consider multi to be a two-element array, with each of these two elements being an array of four integers.

In case this isn't clear to you, Figure 15.2 dissects the array declaration statement into its component parts.

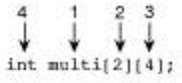

Figure 15-2. The components of a multidimensional array declaration.

Here's how to interpret the components of the declaration:

1. Declare an array named multi.
2. The array multi contains two elements.
3. Each of these two elements contains four elements.
4. Each of the four elements is a type int.

You read a multidimensional array declaration starting with the array name and moving to the right, one set of brackets at a time. When the last set of brackets (the last dimension) has been read, you jump to the beginning of the declaration to determine the array's basic data type.

Under the array-of-arrays scheme, you can visualize a multidimensional array as shown in Figure 15.3.

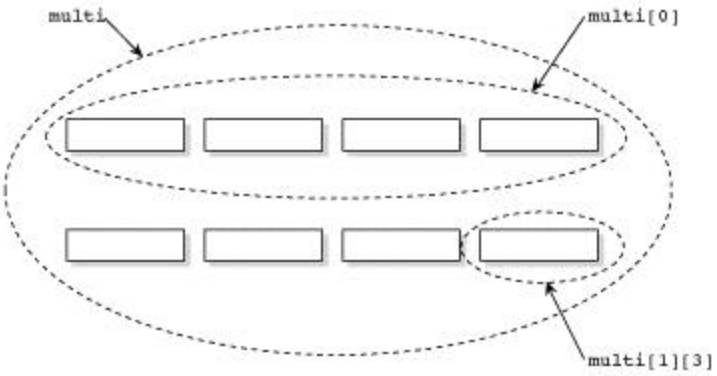

Figure 15-3. A multidimensional array can be visualized as an array of arrays.

Now, let's get back to the topic of array names as pointers. (This is a chapter about pointers, after all!) As with a one-dimensional array, the name of a multidimensional array is a pointer to the first array element. Continuing with our example, multi is a pointer to the first element of the two-dimensional array that was declared as int multi[2][4]. What exactly is the first element of multi? It isn't the type int variable multi[0][0], as you might think. Remember that multi is an array of arrays, so its first element is multi[0], which is an array of four type int variables (one of the two such arrays contained in multi).

Now, if multi[0] is also an array, does it point to anything? Yes, indeed! multi[0] points to its first element, multi[0][0]. You might wonder why multi[0] is a pointer. Remember that the name of an array without brackets is a pointer to

the first array element. The term multi[0] is the name of the array multi[0][0] with the last pair of brackets missing, so it qualifies as a pointer.

If you're a bit confused at this point, don't worry. This material is difficult to grasp. It might help if you remember the following rules for any array of *n* dimensions used in code:

- The array name followed by *n* pairs of brackets (each pair containing an appropriate index, of course) evaluates as array data (that is, the data stored in the specified array element).
- The array name followed by fewer than *n* pairs of brackets evaluates as a pointer to an array element.

In the example, therefore, multi evaluates as a pointer, multi[0] evaluates as a pointer, and multi[0][0] evaluates as array data.

Now look at what all these pointers actually point to. Listing 15.1 declares a two-dimensional array--similar to those you've been using in the examples--and then prints the values of the associated pointers. It also prints the address of the first array element.

Listing 15.1. The relationship between a multidimensional array and pointers.

```
1:    /* Demonstrates pointers and multidimensional arrays. */
2:
3:    #include <stdio.h>
4:
5:    int multi[2][4];
6:
7:    main()
8:    {
9:        printf("\nmulti = %u", multi);
10:       printf("\nmulti[0] = %u", multi[0]);
11:       printf("\n&multi[0][0] = %u\n", &multi[0][0]);
13:       return(0);
12:   }
multi = 1328
multi[0] = 1328
&multi[0][0] = 1328
```

ANALYSIS: The actual value might not be 1328 on your system, but all three values will be the same. The address of the array multi is the same as the address of the array multi[0], and both are equal to the address of the first integer in the array, multi[0][0].

If all three of these pointers have the same value, what is the practical difference between them in terms of your program? Remember from Day 9 that the C compiler knows what a pointer points to. To be more exact, the compiler knows the *size* of the item a pointer is pointing to.

What are the sizes of the elements you've been using? Listing 15.2 uses the operator sizeof() to display the sizes, in bytes, of these elements.

Listing 15.2. Determining the sizes of elements.

```
1: /* Demonstrates the sizes of multidimensional array elements. */
2:
3: #include <stdio.h>
4:
5: int multi[2][4];
```

```
6:
7: main()
8: {
9:     printf("\nThe size of multi = %u", sizeof(multi));
10:    printf("\nThe size of multi[0] = %u", sizeof(multi[0]));
11:    printf("\nThe size of multi[0][0] = %u\n",
sizeof(multi[0][0]));
12:    return(0);
13: }
```

The output of this program (assuming that your compiler uses two-byte integers) is as follows:

```
The size of multi = 16
The size of multi[0] = 8
The size of multi[0][0] = 2
```

ANALYSIS: If you're running a 32-bit operating system, such as IBM's OS/2, your output will be 32, 16, and 4. This is because a type int contains four bytes on these systems.

Think about these size values. The array multi contains two arrays, each of which contains four integers. Each integer requires two bytes of storage. With a total of eight integers, the size of 16 bytes makes sense.

Next, multi[0] is an array containing four integers. Each integer takes two bytes, so the size of eight bytes for multi[0] also makes sense.

Finally, multi[0][0] is an integer, so its size is, of course, two bytes.

Now, keeping these sizes in mind, recall the discussion on Day 9 about pointer arithmetic. The C compiler knows the size of the object being pointed to, and pointer arithmetic takes this size into account. When you increment a pointer, its value is increased by the amount needed to make it point to the "next" of whatever it's pointing to. In other words, it's incremented by the size of the object to which it points.

When you apply this to the example, multi is a pointer to a four-element integer array with a size of 8. If you increment multi, its value should increase by 8 (the size of a four-element integer array). If multi points to multi[0], therefore, (multi + 1) should point to multi[1]. Listing 15.3 tests this theory.

Listing 15.3. Pointer arithmetic with multidimensional arrays.

```
1: /*  Demonstrates pointer arithmetic with pointers */
2: /*  to multidimensional arrays. */
3:
4: #include <stdio.h>
5:
6: int multi[2][4];
7:
8: main()
9: {
10:     printf("\nThe value of (multi) = %u", multi);
11:     printf("\nThe value of (multi + 1) = %u", (multi+1));
12:     printf("\nThe address of multi[1] = %u\n", &multi[1]);
13:     return(0);
14: }
The value of (multi) = 1376
```

```
The value of (multi + 1) = 1384
The address of multi[1] = 1384
```

ANALYSIS: The precise values might be different on your system, but the relationships are the same. Incrementing multi by 1 increases its value by 8 (or by 16 on a 32-bit system) and makes it point to the next element of the array, multi[1].

In this example, you've seen that multi is a pointer to multi[0]. You've also seen that multi[0] is itself a pointer (to multi[0][0]). Therefore, multi is a pointer to a pointer. To use the expression multi to access array data, you must use double indirection. To print the value stored in multi[0][0], you could use any of the following three statements:

```
printf("%d", multi[0][0]);
printf("%d", *multi[0]);
printf("%d", **multi);
```

These concepts apply equally to arrays with three or more dimensions. Thus, a three-dimensional array is an array with elements that are each a two-dimensional array; each of these elements is itself an array of one-dimensional arrays.

This material on multidimensional arrays and pointers might seem a bit confusing. When you work with multidimensional arrays, keep this point in mind: An array with *n* dimensions has elements that are arrays with *n*-1 dimensions. When *n* becomes 1, that array's elements are variables of the data type specified at the beginning of the array declaration line.

So far, you've been using array names that are pointer constants and that can't be changed. How would you declare a pointer variable that points to an element of a multidimensional array? Let's continue with the previous example, which declared a two-dimensional array as follows:

```
int multi[2][4];
```

To declare a pointer variable ptr that can point to an element of multi (that is, can point to a four-element integer array), you would write

```
int (*ptr)[4];
```

You could then make ptr point to the first element of multi by writing

```
ptr = multi;
```

You might wonder why the parentheses are necessary in the pointer declaration. Brackets ([ ]) have a higher precedence than *. If you wrote

```
int *ptr[4];
```

you would be declaring an array of four pointers to type int. Indeed, you can declare and use arrays of pointers. This isn't what you want to do now, however.

How can you use pointers to elements of multidimensional arrays? As with single-dimensional arrays, pointers must be used to pass an array to a function. This is illustrated for a multidimensional array in Listing 15.4, which uses two methods of passing a multidimensional array to a function.

Listing 15.4. Passing a multidimensional array to a function using a pointer.

```
1:  /* Demonstrates passing a pointer to a multidimensional */
2:  /* array to a function. */
```

```
3:
4:  #include <stdio.h>
5:
6:  void printarray_1(int (*ptr)[4]);
7:  void printarray_2(int (*ptr)[4], int n);
8:
9:  main()
10: {
11:    int  multi[3][4] = { { 1, 2, 3, 4 },
12:                         { 5, 6, 7, 8 },
13:                         { 9, 10, 11, 12 } };
14:
15:    /* ptr is a pointer to an array of 4 ints. */
16:
17:    int (*ptr)[4], count;
18:
19:    /* Set ptr to point to the first element of multi. */
20:
21:    ptr = multi;
22:
23:    /* With each loop, ptr is incremented to point to the next */
24:    /* element (that is, the next 4-element integer array) of
multi. */
25:
26:    for (count = 0; count < 3; count++)
27:        printarray_1(ptr++);
28:
29:    puts("\n\nPress Enter...");
30:    getchar();
31:    printarray_2(multi, 3);
32:    printf("\n");
33:    return(0);
34: }
35
36: void printarray_1(int (*ptr)[4])
37: {
38: /* Prints the elements of a single four-element integer array. */
39: /* p is a pointer to type int. You must use a type cast */
40: /* to make p equal to the address in ptr. */
41:
42:      int *p, count;
43:      p = (int *)ptr;
44:
45:      for (count = 0; count < 4; count++)
46:          printf("\n%d", *p++);
47: }
48:
49: void printarray_2(int (*ptr)[4], int n)
50: {
51: /* Prints the elements of an n by four-element integer array. */
52:
```

```
53:      int *p, count;
54:      p = (int *)ptr;
55:
56:      for (count = 0; count < (4 * n); count++)
57:          printf("\n%d", *p++);
58: }
1
2
3
4
5
6
7
8
9
10
11
12
Press Enter...
1
2
3
4
5
6
7
8
9
10
11
12
```

ANALYSIS: On lines 11 through 13, the program declares and initializes an array of integers, multi[3][4]. Lines 6 and 7 are the prototypes for the functions printarray_1() and printarray_2(), which print the contents of the array.

The function printarray_1() (lines 36 through 47) is passed only one argument, a pointer to an array of four integers. This function prints all four elements of the array. The first time main() calls printarray_1() on line 27, it passes a pointer to the first element (the first four-element integer array) in multi. It then calls the function two more times, incrementing the pointer each time to point to the second, and then to the third, element of multi. After all three calls are made, the 12 integers in multi are displayed on-screen.

The second function, printarray_2(), takes a different approach. It too is passed a pointer to an array of four integers, but, in addition, it is passed an integer variable that specifies the number of elements (the number of arrays of four integers) that the multidimensional array contains. With a single call from line 31, printarray_2() displays the entire contents of multi.

Both functions use pointer notation to step through the individual integers in the array. The notation (int *)ptr in both functions (lines 43 and 54) might not be clear. The (int *) is a typecast, which temporarily changes the variable's data type from its declared data type to a new one. The typecast is required when assigning the value of ptr to p because they are pointers to different types (p is a pointer to type int, whereas ptr is a pointer to an array of four integers). C doesn't let you assign the value of one pointer to a pointer of a different type. The typecast tells the compiler, "For this statement only, treat ptr as a pointer to type int." Day 20, "Working with Memory," covers typecasts in more detail.

DON'T forget to use the double indirection operator (**) when declaring a pointer to a pointer.

DON'T forget that a pointer increments by the size of the pointer's type (usually what is being pointed to).

DON'T forget to use parentheses when declaring pointers to arrays.
To declare a pointer to an array of characters, use this format:

```
char (*letters)[26];
```

To declare an array of pointers to characters, use this format:

```
char *letters[26];
```

15-3. Arrays of Pointers

Recall from Day 8, "Using Numeric Arrays," that an array is a collection of data storage locations that have the same data type and are referred to by the same name. Because pointers are one of C's data types, you can declare and use arrays of pointers. This type of program construct can be very powerful in certain situations.

Perhaps the most common use of an array of pointers is with strings. A string, as you learned on Day 10, "Characters and Strings," is a sequence of characters stored in memory. The start of the string is indicated by a pointer to the first character (a pointer to type char), and the end of the string is marked by a null character. By declaring and initializing an array of pointers to type char, you can access and manipulate a large number of strings using the pointer array. Each element in the array points to a different string, and by looping through the array, you can access each of them in turn.

15-3-1. Strings and Pointers: A Review

This is a good time to review some material from Day 10 regarding string allocation and initialization. One way to allocate and initialize a string is to declare an array of type char as follows:

```
char message[] = "This is the message.";
```

You could accomplish the same thing by declaring a pointer to type char:

```
char *message = "This is the message.";
```

Both declarations are equivalent. In each case, the compiler allocates enough space to hold the string along with its terminating null character, and the expression message is a pointer to the start of the string. But what about the following two declarations?

```
char message1[20];
char *message2;
```

The first line declares an array of type char that is 20 characters long, with message1 being a pointer to the first array position. Although the array space is allocated, it isn't initialized, and the array contents are undetermined. The second line declares message2, a pointer to type char. No storage space for a string is allocated by this statement--only space to hold the pointer. If you want to create a string and then have message2 point to it, you must allocate space for the string first. On Day 10, you learned how to use the malloc() memory allocation function

for this purpose. Remember that any string must have space allocated for it, whether at compilation in a declaration or at runtime with malloc() or a related memory allocation function.

15-3-2. Array of Pointers to Type char

Now that you're done with the review, how do you declare an array of pointers? The following statement declares an array of 10 pointers to type char:

```
char *message[10];
```

Each element of the array message[] is an individual pointer to type char. As you might have guessed, you can combine the declaration with initialization and allocation of storage space for the strings:

```
char *message[10] = { "one", "two", "three" };
```

This declaration does the following:

- It allocates a 10-element array named message; each element of message is a pointer to type char.
- It allocates space somewhere in memory (exactly where doesn't concern you) and stores the three initialization strings, each with a terminating null character.
- It initializes message[0] to point to the first character of the string "one", message[1] to point to the first character of the string "two", and message[2] to point to the first character of the string "three".

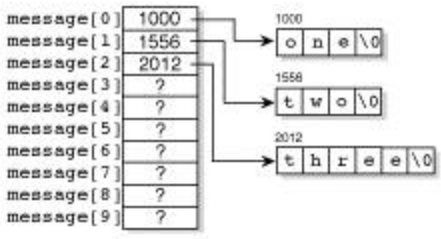

Figure 15-4. An array of pointers to type char.

This is illustrated in Figure 15.4, which shows the relationship between the array of pointers and the strings. Note that in this example, the array elements message[3] through message[9] aren't initialized to point at anything.

Now look at Listing 15.5, which is an example of using an array of pointers.

Listing 15.5. Initializing and using an array of pointers to type char.

```
1: /* Initializing an array of pointers to type char. */
2:
3: #include <stdio.h>
4:
5: main()
6: {
7:     char *message[8] = { "Four", "score", "and", "seven",
8:                          "years", "ago,", "our", "forefathers" };
9:   int count;
10:
11:     for (count = 0; count < 8; count++)
12:         printf("%s ", message[count]);
13:     printf("\n");
14:     return(0);
15: }
```

```
Four score and seven years ago, our forefathers
```

ANALYSIS: This program declares an array of eight pointers to type char and initializes them to point to eight strings (lines 7 and 8). It then uses a for loop on lines 11 and 12 to display each element of the array on-screen.

You probably can see how manipulating the array of pointers is easier than manipulating the strings themselves. This advantage is obvious in more complicated programs, such as the one presented later in this chapter. As you'll see in that program, the advantage is greatest when you're using functions. It's much easier to pass an array of pointers to a function than to pass several strings. This can be illustrated by rewriting the program in Listing 15.5 so that it uses a function to display the strings. The modified program is shown in Listing 15.6.

Listing 15.6. Passing an array of pointers to a function.

```
1: /* Passing an array of pointers to a function. */
2:
3: #include <stdio.h>
4:
5: void print_strings(char *p[], int n);
6:
7: main()
8: {
9:     char *message[8] = { "Four", "score", "and", "seven",
10:                         "years", "ago,", "our", "forefathers" };
11:
12:     print_strings(message, 8);
13:     return(0);
14: }
15:
16: void print_strings(char *p[], int n)
17: {
18:     int count;
19:
20:     for (count = 0; count < n; count++)
21:         printf("%s ", p[count]);
22:    printf("\n");
23:  }
```

```
Four score and seven years ago, our forefathers
```

ANALYSIS: Looking at line 16, you see that the function print_strings() takes two arguments. One is an array of pointers to type char, and the other is the number of elements in the array. Thus, print_strings() could be used to print the strings pointed to by any array of pointers.

You might remember that, in the section on pointers to pointers, you were told that you would see a demonstration later. Well, you've just seen it. Listing 15.6 declared an array of pointers, and the name of the array is a pointer to its first element. When you pass that array to a function, you're passing a pointer (the array name) to a pointer (the first array element).

15-3-3. An Example

Now it's time for a more complicated example. Listing 15.7 uses many of the programming skills you've learned, including arrays of pointers. This program accepts lines of input from the keyboard, allocating space for each line

as it is entered and keeping track of the lines by means of an array of pointers to type char. When you signal the end of an entry by entering a blank line, the program sorts the strings alphabetically and displays them on-screen.

If you were writing this program from scratch, you would approach the design of this program from a structured programming perspective. First, make a list of the things the program must do:

1. Accept lines of input from the keyboard one at a time until a blank line is entered.
2. Sort the lines of text into alphabetical order.
3. Display the sorted lines on-screen.

This list suggests that the program should have at least three functions: one to accept input, one to sort the lines, and one to display the lines. Now you can design each function independently. What do you need the input function--called get_lines()--to do? Again, make a list:

1. Keep track of the number of lines entered, and return that value to the calling program after all lines have been entered.
2. Don't allow input of more than a preset maximum number of lines.
3. Allocate storage space for each line.
4. Keep track of all lines by storing pointers to strings in an array.
5. Return to the calling program when a blank line is entered.

Now think about the second function, the one that sorts the lines. It could be called sort(). (Really original, right?) The sort technique used is a simple, brute-force method that compares adjacent strings and swaps them if the second string is less than the first string. More exactly, the function compares the two strings whose pointers are adjacent in the array of pointers and swaps the two pointers if necessary.

To be sure that the sorting is complete, you must go through the array from start to finish, comparing each pair of strings and swapping if necessary. For an array of *n* elements, you must go through the array *n*-1 times. Why is this necessary?

Each time you go through the array, a given element can be shifted by, at most, one position. For example, if the string that should be first is actually in the last position, the first pass through the array moves it to the next-to-last position, the second pass through the array moves it up one more position, and so on. It requires *n*-1 passes to move it to the first position, where it belongs.

Note that this is a very inefficient and inelegant sorting method. However, it's easy to implement and understand, and it's more than adequate for the short lists that the sample program sorts.

The final function displays the sorted lines on-screen. It is, in effect, already written in List- ing 15.6, and it requires only minor modification for use in Listing 15.7.

Listing 15.7. A program that reads lines of text from the keyboard, sorts them alphabetically, and displays the sorted list.

```
1:  /* Inputs a list of strings from the keyboard, sorts them, */
2:  /* and then displays them on the screen. */
3:  #include <stdlib.h>
4:  #include <stdio.h>
5:  #include <string.h>
6:
7:  #define MAXLINES 25
8:
9:  int get_lines(char *lines[]);
10: void sort(char *p[], int n);
11: void print_strings(char *p[], int n);
12:
```

```
13: char *lines[MAXLINES];
14:
15: main()
16: {
17:     int number_of_lines;
18:
19:     /* Read in the lines from the keyboard. */
20:
21:     number_of_lines = get_lines(lines);
22:
23:     if ( number_of_lines < 0 )
24:     {
25:         puts(" Memory allocation error");
26:         exit(-1);
27:     }
28:
29:     sort(lines, number_of_lines);
30:     print_strings(lines, number_of_lines);
31:     return(0);
32: }
33:
34: int get_lines(char *lines[])
35: {
36:     int n = 0;
37:     char buffer[80];  /* Temporary storage for each line. */
38:
39:     puts("Enter one line at time; enter a blank when done.");
40:
41:     while ((n < MAXLINES) && (gets(buffer) != 0) &&
42:            (buffer[0] != `\0'))
43:     {
44:         if ((lines[n] = (char *)malloc(strlen(buffer)+1)) == NULL)
45:             return -1;
46:         strcpy( lines[n++], buffer );
47:     }
48:     return n;
49:
50: } /* End of get_lines() */
51:
52: void sort(char *p[], int n)
53: {
54:     int a, b;
55:     char *x;
56:
57:     for (a = 1; a < n; a++)
58:     {
59:         for (b = 0; b < n-1; b++)
60:         {
61:              if (strcmp(p[b], p[b+1]) > 0)
62:             {
63:                  x = p[b];
```

```
64:                     p[b] = p[b+1];
65:                     p[b+1] = x;
66:                 }
67:             }
68:         }
69: }
70:
71: void print_strings(char *p[], int n)
72: {
73:     int count;
74:
75:     for (count = 0; count < n; count++)
76:         printf("%s\n ", p[count]);
77: }
Enter one line at time; enter a blank when done.
dog
apple
zoo
program
merry
apple
dog
merry
program
zoo
```

ANALYSIS: It will be worthwhile for you to examine some of the details of this program. Several new library functions are used for various types of string manipulation. They are explained briefly here and in more detail on Day 17, "Manipulating Strings." The header file STRING.H must be included in a program that uses these functions.

In the get_lines() function, input is controlled by the while statement on lines 41 and 42, which read as follows (condensed here onto one line):

```
while ((n < MAXLINES) && (gets(buffer) != 0) && (buffer[0] != `\0'))
```

The condition tested by the while has three parts. The first part, n < MAXLINES, ensures that the maximum number of lines has not been input yet. The second part, gets(buffer) != 0, calls the gets() library function to read a line from the keyboard into buffer and verifies that end-of-file or some other error has not occurred. The third part, buffer[0] != `\0', verifies that the first character of the line just input is not the null character, which would signal that a blank line had been entered.

If any of these three conditions isn't satisfied, the while loop terminates, and execution returns to the calling program, with the number of lines entered as the return value. If all three conditions are satisfied, the following if statement on line 44 is executed:

```
if ((lines[n] = (char *)malloc(strlen(buffer)+1)) == NULL)
```

This statement calls malloc() to allocate space for the string that was just input. The strlen() function returns the length of the string passed as an argument; the value is incremented by 1 so that malloc() allocates space for the string plus its terminating null character.

The library function malloc(), you might remember, returns a pointer. The statement assigns the value of the pointer returned by malloc() to the corresponding element of the array of pointers. If malloc() returns NULL, the if loop

returns execution to the calling program with a return value of -1. The code in main() tests the return value of get_lines() and checks whether a value less than 0 is returned; lines 23 through 27 report a memory allocation error and terminate the program.

If the memory allocation was successful, the program uses the strcpy() function on line 46 to copy the string from the temporary storage location buffer to the storage space just allocated by malloc(). The while loop then repeats, getting another line of input.

Once execution returns from get_lines() to main(), the following has been accomplished (assuming that a memory allocation error didn't occur):

- A number of lines of text have been read from the keyboard and stored in memory as null-terminated strings.
- The array lines[] contains pointers to all the strings. The order of pointers in the array is the order in which the strings were input.
- The variable number_of_lines holds the number of lines that were input.

Now it's time to sort. Remember, you're not actually going to move the strings around, only the order of the pointers in the array lines[]. Look at the code in the function sort(). It contains one for loop nested inside another (lines 57 through 68). The outer loop executes number_of_lines - 1 times. Each time the outer loop executes, the inner loop steps through the array of pointers, comparing (string n) with (string n+1) for n = 0 to n = number_of_lines - 1. The comparison is performed by the library function strcmp() on line 61, which is passed pointers to two strings. The function strcmp() returns one of the following:

- A value greater than zero if the first string is greater than the second string.
- Zero if the two strings are identical.
- A value less than zero if the second string is greater than the first string.

In the program, a return value from strcmp() that is greater than zero means that the first string is "greater than" the second string, and they must be swapped (that is, their pointers in lines[] must be swapped). This is done using a temporary variable x. Lines 63 through 65 perform the swap.

When program execution returns from sort(), the pointers in lines[] are ordered properly: A pointer to the "lowest" string is in lines[0], a pointer to the "next-lowest" is in lines[1], and so on. Suppose, for example, that you entered the following five lines, in this order:

```
dog
apple
zoo
program
merry
```

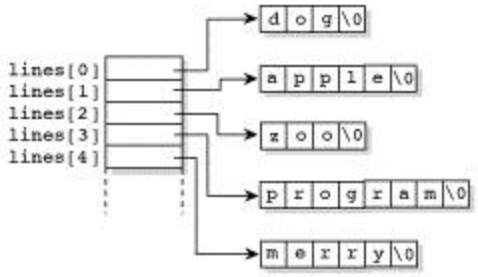

Figure 5. Before sorting, the pointers are in the same order in which the strings were entered.

The situation before calling sort() is illustrated in Figure 15.5, and the situation after the return from sort() is illustrated in Figure 15.6 (not showing in any figure).

Finally, the program calls the function print_strings() to display the sorted list of strings on-screen. This function should be familiar to you from previous examples in this chapter.

The program in Listing 15.7 is the most complex you have yet encountered in this book. It uses many of the C programming techniques that were covered in previous chapters. With the aid of the preceding explanation, you should be able to follow the program's operation and understand each step. If you find areas that are unclear to you, review the related sections of this book until you understand.

15-4. Pointers to Functions

Pointers to functions provide another way of calling functions. "Hold on," you might be thinking. "How can you have a pointer to a function? Doesn't a pointer hold the address where a variable is stored?"

Well, yes and no. It's true that a pointer holds an address, but it doesn't have to be the address where a variable is stored. When your program runs, the code for each function is loaded into memory starting at a specific address. A pointer to a function holds the starting address of a function--its entry point.

Why use a pointer to a function? As I mentioned earlier, it provides a more flexible way of calling a function. It lets the program choose from among several functions, selecting the one that is appropriate for the current circumstances.

15-4-1. Declaring a Pointer to a Function

Like other pointers, you must declare a pointer to a function. The general form of the declaration is as follows:

```
type (*ptr_to_func)(parameter_list);
```

This statement declares ptr_to_func as a pointer to a function that returns *type* and is passed the parameters in *parameter_list*. Here are some more concrete examples:

```
int (*func1)(int x);
void (*func2)(double y, double z);
char (*func3)(char *p[]);
void (*func4)();
```

The first line declares func1 as a pointer to a function that takes one type int argument and returns a type int. The second line declares func2 as a pointer to a function that takes two type double arguments and has a void return type (no return value). The third line declares func3 as a pointer to a function that takes an array of pointers to type char as its argument and returns a type char. The final line declares func4 as a pointer to a function that doesn't take any arguments and has a void return type.

Why do you need parentheses around the pointer name? Why can't you write, for the first example:

```
int *func1(int x);
```

The reason has to do with the precedence of the indirection operator, *. It has a relatively low precedence, lower than the parentheses surrounding the parameter list. The declaration just given, without the first set of parentheses, declares func1 as a function that returns a pointer to type int. (Functions that return pointers are covered on Day 18, "Getting More from Functions.") When you declare a pointer to a function, always remember to include a set of parentheses around the pointer name and indirection operator, or you will get into trouble.

15-4-2. Initializing and Using a Pointer to a Function

A pointer to a function must not only be declared, but also initialized to point to something. That "something" is, of course, a function. There's nothing special about a function that gets pointed to. The only requirement is that its return type and parameter list match the return type and parameter list of the pointer declaration. For example, the following code declares and defines a function and a pointer to that function:

```
float square(float x);     /* The function prototype.  */
float (*p)(float x);       /* The pointer declaration. */
float square(float x)      /* The function definition. */
{
return x * x;
}
```

Because the function square() and the pointer p have the same parameter and return types, you can initialize p to point to square as follows:

```
p = square;
```

Then you can call the function using the pointer as follows:

```
answer = p(x);
```

It's that simple. For a real example, compile and run Listing 15.8, which declares and initializes a pointer to a function and then calls the function twice, using the function name the first time and the pointer the second time. Both calls produce the same result.

Listing 15.8. Using a pointer to a function to call the function.

```
1:  /* Demonstration of declaring and using a pointer to a function.*/
2:
3:  #include <stdio.h>
4:
5:  /* The function prototype. */
6:
7:  double square(double x);
8:
9:  /* The pointer declaration. */
10:
11: double (*p)(double x);
12:
13: main()
14: {
15:     /* Initialize p to point to square(). */
16:
17:     p = square;
18:
19:     /* Call square() two ways. */
20:     printf("%f  %f\n", square(6.6), p(6.6));
21:     return(0);
22: }
23:
24: double square(double x)
25: {
26:     return x * x;
27: }
43.559999  43.559999
```

NOTE: Precision of the values might cause some numbers to not display as the exact values entered. For example, the correct answer, 43.56, might appear as 43.559999.

Line 7 declares the function square(), and line 11 declares the pointer p to a function containing a double argument and returning a double value, matching the declaration of square(). Line 17 sets the pointer p equal to square. Notice that parentheses aren't used with square or p. Line 20 prints the return values from calls to square() and p().

A function name without parentheses is a pointer to the function (sounds similar to the situation with arrays, doesn't it?). What's the point of declaring and using a separate pointer to the function? Well, the function name itself is a pointer constant and can't be changed (again, a parallel to arrays). A pointer variable, in contrast, *can* be changed. Specifically, it can be made to point to different functions as the need arises.

Listing 15.9 calls a function, passing it an integer argument. Depending on the value of the argument, the function initializes a pointer to point to one of three other functions and then uses the pointer to call the corresponding function. Each of these three functions displays a specific message on-screen.

Listing 15.9. Using a pointer to a function to call different functions depending on program circumstances.

```
1:  /* Using a pointer to call different functions. */
2:
3:  #include <stdio.h>
4:
5:  /* The function prototypes. */
6:
7:  void func1(int x);
8:  void one(void);
9:  void two(void);
10: void other(void);
11:
12: main()
13: {
14:      int a;
15:
16:     for (;;)
17:     {
18:         puts("\nEnter an integer between 1 and 10, 0 to exit: ");
19:          scanf("%d", &a);
20:
21:          if (a == 0)
22:             break;
23:         func1(a);
24:     }
25:     return(0);
26: }
27:
28: void func1(int x)
29: {
30:     /* The pointer to function. */
31:
32:     void (*ptr)(void);
33:
34:     if (x == 1)
35:         ptr = one;
36:     else if (x == 2)
37:         ptr = two;
```

```
38:     else
39:         ptr = other;
40:
41:     ptr();
42: }
43:
44: void one(void)
45: {
46:     puts("You entered 1.");
47: }
48:
49: void two(void)
50: {
51:     puts("You entered 2.");
52: }
53:
54: void other(void)
55: {
56:     puts("You entered something other than 1 or 2.");
57: }
Enter an integer between 1 and 10, 0 to exit:
2
You entered 2.
Enter an integer between 1 and 10, 0 to exit:
9
You entered something other than 1 or 2.
Enter an integer between 1 and 10, 0 to exit:
0
```

ANALYSIS: This program employs an infinite loop starting on line 16 to continue execution until a value of 0 is entered. When a nonzero value is entered, it's passed to func1(). Note that line 32, in func1(), contains a declaration for a pointer ptr to a function. Being declared within a function makes ptr local to func1(), which is appropriate because no other part of the program needs access to it. func1() then uses this value to set ptr equal to the appropriate function (lines 34 through 39). Line 41 then makes a single call to ptr(), which calls the appropriate function.

Of course, this program is for illustration purposes only. You could have easily accomplished the same result without using a pointer to a function.

Now you can learn another way to use pointers to call different functions: passing the pointer as an argument to a function. Listing 15.10 is a revision of Listing 15.9.

Listing 15.10. Passing a pointer to a function as an argument.

```
1:  /* Passing a pointer to a function as an argument. */
2:
3:  #include <stdio.h>
4:
5:  /* The function prototypes. The function func1() takes as */
6:  /* its one argument a pointer to a function that takes no */
7:  /* arguments and has no return value. */
8:
9:  void func1(void (*p)(void));
```

```
10: void one(void);
11: void two(void);
12: void other(void);
13:
14: main()
15: {
16:     /* The pointer to a function. */
18:     void (*ptr)(void);
19:     int  a;
20:
21:     for (;;)
22:     {
23:         puts("\nEnter an integer between 1 and 10, 0 to exit: ");
24:         scanf("%d", &a);
25:
26:         if (a == 0)
27:             break;
28:         else if (a == 1)
29:             ptr = one;
30:         else if (a == 2)
31:             ptr = two;
32:         else
33:             ptr = other;
34:         func1(ptr);
35:     }
36:     return(0);
37: }
38:
39: void func1(void (*p)(void))
40: {
41:     p();
42: }
43:
44: void one(void)
45: {
46:     puts("You entered 1.");
47: }
48:
49: void two(void)
50: {
51:     puts("You entered 2.");
52: }
53:
54: void other(void)
55: {
56:     puts("You entered something other than 1 or 2.");
57: }
Enter an integer between 1 and 10, 0 to exit:
2
You entered 2.
Enter an integer between 1 and 10, 0 to exit:
```

```
11
You entered something other than 1 or 2.
Enter an integer between 1 and 10, 0 to exit:
0
```

ANALYSIS: Notice the differences between Listing 15.9 and Listing 15.10. The declaration of the pointer to a function has been moved to line 18 in main(), where it is needed. Code in main() now initializes the pointer to point to the correct function, depending on the value the user entered (lines 26 through 33), and then passes the initialized pointer to func1(). func1() really serves no purpose in Listing 15.10; all it does is call the function pointed to by ptr. Again, this program is for illustration purposes. The same principles can be used in real-world programs, such as the example in the next section.

One programming situation in which you might use pointers to functions is when sorting is required. Sometimes you might want different sorting rules used. For example, you might want to sort in alphabetical order one time and in reverse alphabetical order another time. By using pointers to functions, your program can call the correct sorting function. More precisely, it's usually a different comparison function that's called.

Look back at Listing 15.7. In the sort() function, the actual sort order is determined by the value returned by the strcmp() library function, which tells the program whether a given string is "less than" or "greater than" another string. What if you wrote two comparison functions--one that sorts alphabetically (where A is less than Z), and another that sorts in reverse alphabetical order (where Z is less than A)? The program can ask the user what order he wants and, by using pointers, the sorting function can call the proper comparison function. Listing 15.11 modifies Listing 15.7 to incorporate this feature.

Listing 15.11. Using pointers to functions to control sort order.

```
1:  /* Inputs a list of strings from the keyboard, sorts them */
2:  /* in ascending or descending order, and then displays them */
3:  /* on the screen. */
4:  #include <stdlib.h>
5:  #include <stdio.h>
6:  #include <string.h>
7:
8:  #define MAXLINES 25
9:
10:  int get_lines(char *lines[]);
11:  void sort(char *p[], int n, int sort_type);
12:  void print_strings(char *p[], int n);
13:  int alpha(char *p1, char *p2);
14:  int reverse(char *p1, char *p2);
15:
16:  char *lines[MAXLINES];
17:
18:  main()
19:  {
20:     int number_of_lines, sort_type;
21:
22:     /* Read in the lines from the keyboard. */
23:
24:     number_of_lines = get_lines(lines);
25:
26:     if ( number_of_lines < 0 )
27:     {
28:        puts("Memory allocation error");
```

```
29:        exit(-1);
30:     }
31:
32:     puts("Enter 0 for reverse order sort, 1 for alphabetical:" );
33:     scanf("%d", &sort_type);
34:
35:     sort(lines, number_of_lines, sort_type);
36:     print_strings(lines, number_of_lines);
37:     return(0);
38:  }
39:
40:  int get_lines(char *lines[])
41:  {
42:      int n = 0;
43:      char buffer[80];  /* Temporary storage for each line. */
44:
45:      puts("Enter one line at time; enter a blank when done.");
46:
47:      while (n < MAXLINES && gets(buffer) != 0 && buffer[0] !=
`\0')
48:      {
49:          if ((lines[n] = (char *)malloc(strlen(buffer)+1)) ==
NULL)
50:          return -1;
51:          strcpy( lines[n++], buffer );
52:      }
53:      return n;
54:
55:  } /* End of get_lines() */
56:
57:  void sort(char *p[], int n, int sort_type)
58:  {
59:      int a, b;
60:      char *x;
61:
62:      /* The pointer to function.  */
63:
64:      int (*compare)(char *s1, char *s2);
65:
66:      /* Initialize the pointer to point to the proper comparison
*/
67:      /* function depending on the argument sort_type. */
68:
69:      compare = (sort_type) ? reverse : alpha;
70:
71:      for (a = 1; a < n; a++)
72:      {
73:          for (b = 0; b < n-1; b++)
74:          {
75:              if (compare(p[b], p[b+1]) > 0)
76:              {
```

```
77:                 x = p[b];
78:                 p[b] = p[b+1];
79:                 p[b+1] = x;
80:             }
81:         }
82:     }
83: }  /* end of sort() */
84:
85: void print_strings(char *p[], int n)
86: {
87:     int count;
88:
89:     for (count = 0; count < n; count++)
90:         printf("%s\n ", p[count]);
91: }
92:
93: int alpha(char *p1, char *p2)
94: /* Alphabetical comparison. */
95: {
96:     return(strcmp(p2, p1));
97: }
98:
99: int reverse(char *p1, char *p2)
100: /* Reverse alphabetical comparison. */
101: {
102:      return(strcmp(p1, p2));
103: }
Enter one line at time; enter a blank when done.
Roses are red
Violets are blue
C has been around,
But it is new to you!
Enter 0 for reverse order sort, 1 for alphabetical:
0
Violets are blue
Roses are red
C has been around,
But it is new to you!
```

ANALYSIS: Lines 32 and 33 in main() prompt the user for the desired sort order. The value entered is placed in sort_type. This value is passed to the sort() function along with the other information described for Listing 15.7. The sort() function contains a couple of changes. Line 64 declares a pointer to a function called compare() that takes two character pointers (strings) as arguments. Line 69 sets compare() equal to one of the two new functions added to the listing based on the value of sort_type. The two new functions are alpha() and reverse(). alpha() uses the strcmp() library function just as it was used in Listing 15.7; reverse() does not. reverse() switches the parameters passed so that a reverse-order sort is done.

DO use structured programming.

DON'T forget to use parentheses when declaring pointers to functions.
Here's how you declare a pointer to a function that takes no arguments and returns a character:

```
char (*func)();
```

Here's how you declare a function that returns a pointer to a character:

```
char *func();
```

DO initialize a pointer before using it.

DON'T use a function pointer that has been declared with a different return type or different arguments than what you need.

15-5. Linked Lists

A *linked list* is a useful method of data storage that can easily be implemented in C. Why are we covering linked lists in a chapter on pointers? Because, as you will soon see, pointers are central to linked lists.

There are several kinds of linked lists, including single-linked lists, double-linked lists, and binary trees. Each type is suited for certain types of data storage. The one thing that these lists have in common is that the links between data items are defined by information that is contained in the items themselves, in the form of pointers. This is distinctly different from arrays, in which the links between data items result from the layout and storage of the array. This section explains the most fundamental kind of linked list: the single-linked list (which I refer to as simply a linked list).

15-5-1. Basics of Linked Lists

Each data item in a linked list is contained in a structure (you learned about structures on Day 11, "Structures"). The structure contains the data elements needed to hold the data being stored; these depend on the needs of the specific program. In addition, there is one more data element--a pointer. This pointer provides the links in a linked list. Here's a simple example:

```
struct person {
char name[20];
struct person *next;
};
```

This code defines a structure named person. For the data, person contains only a 20-element array of characters. You generally wouldn't use a linked list for such simple data, but this will serve for an example. The person structure also contains a pointer to type person--in other words, a pointer to another structure of the same type. This means that each structure of type person can not only contain a chunk of data, but also can point to another person structure. Figure 15.6 shows how this lets the structures be linked together in a list.

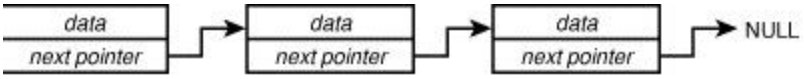

Figure 15-6. Links in a linked list.

Notice that in Figure 15.6, each person structure points to the next person structure. The last person structure doesn't point to anything. The last element in a linked list is identified by the pointer element being assigned the value of NULL.

NOTE: The structures that make up a link in a linked list can be referred to as *links, nodes,* or *elements* of a linked list.

You have seen how the last link in a linked list is identified. What about the first link? This is identified by a special pointer (not a structure) called the *head pointer*. The head pointer always points to the first element in the linked list. The first element contains a pointer to the second element, the second element contains a pointer to the third, and so on until you encounter an element whose pointer is NULL. If the entire list is empty (contains no links), the head pointer is set to NULL. Figure 15.7 illustrates the head pointer before the list is started and after the first list element is added.

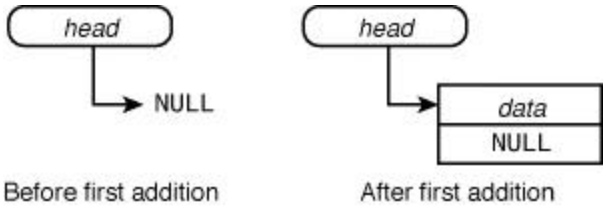

Figure 15-7. A linked list's head pointer.

NOTE: The *head pointer* is a pointer to the first element in a linked list. The head pointer is sometimes referred to as the *first element pointer* or *top pointer*.

15-5-2. Working with Linked Lists

When you're working with a linked list, you can add, delete, or modify elements or links. Modifying an element presents no real challenge; however, adding and deleting elements can. As I stated earlier, elements in a list are connected with pointers. Much of the work of adding and deleting elements consists of manipulating these pointers. Elements can be added to the beginning, middle, or end of a linked list; this determines how the pointers must be changed.

Later in this chapter you'll find a simple linked list demonstration, as well as a more complex working program. Before getting into the nitty-gritty of code, however, it's a good idea to examine some of the actions you need to perform with linked lists. For these sections, we will continue using the person structure that was introduced earlier.

Preliminaries

Before you can start a linked list, you need to define the data structure that will be used for the list, and you also need to declare the head pointer. Since the list starts out empty, the head pointer should be initialized to NULL. You will also need an additional pointer to your list structure type for use in adding records (you might need more than one pointer, as you'll soon see). Here's how you do it:

```
struct person {
  char name[20];
  struct person *next;
};
struct person *new;
struct person *head;
head = NULL;
```

Adding an Element to the Beginning of a List

If the head pointer is NULL, the list is empty, and the new element will be its only member. If the head pointer is not NULL, the list already contains one or more elements. In either case, however, the procedure for adding a new element to the start of the list is the same:

1. Create an instance of your structure, allocating memory space using malloc().
2. Set the next pointer of the new element to the current value of the head pointer. This will be NULL if the list is empty, or the address of the current first element otherwise.
3. Make the head pointer point to the new element.

Here is the code to perform this task:

```
new = (person*)malloc(sizeof(struct person));
new->next = head;
head = new
```

WARNING: It's important to switch the pointers in the correct order. If you reassign the head pointer first, you will lose the list!

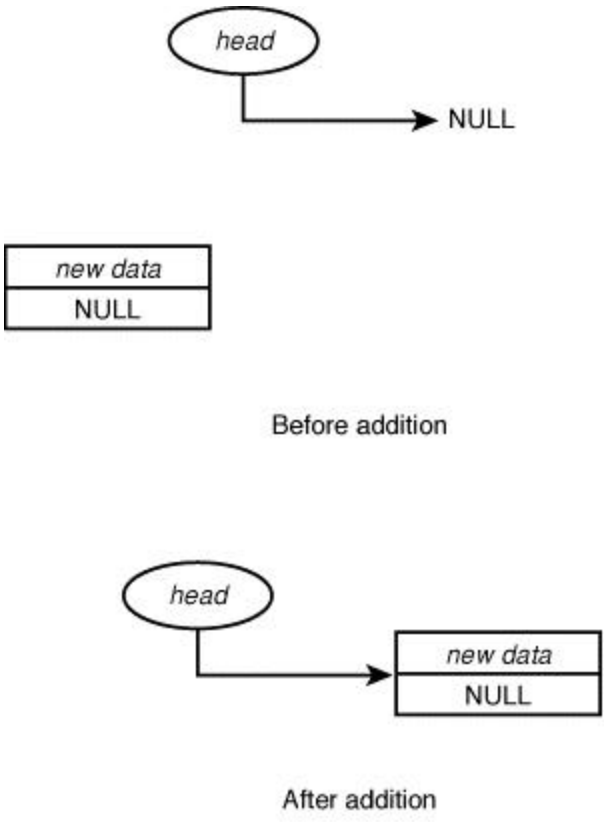

Figure 15-8. Adding a new element to an empty linked list.

Figure 15.8 illustrates the procedure for adding a new element to an empty list, and Figure 15.9 illustrates adding a new first element to an existing list.

Notice that malloc() is used to allocate the memory for the new element. As each new element is added, only the memory needed for it is allocated. The calloc() function could also be used. You should be aware of the differences between these two functions. The main difference is that calloc() will initialize the new element; malloc() will not.

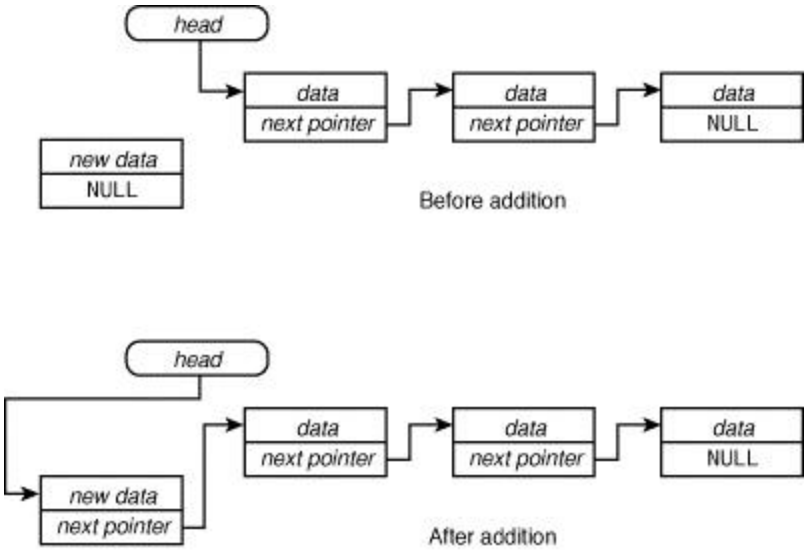

Figure 15-9. Adding a new first element to an existing list.

WARNING: The malloc() in the preceding code fragment didn't ensure that the memory was allocated. You should always check the return value of a memory allocation function.

TIP: When possible, initialize pointers to NULL when you declare them. Never leave a pointer uninitialized.

Adding an Element to the End of the List

To add an element to the end of a linked list, you need to start at the head pointer and go through the list until you find the last element. After you've found it, follow these steps:

1. Create an instance of your structure, allocating memory space using malloc().
2. Set the next pointer in the last element to point to the new element (whose address is returned by malloc()).
3. Set the next pointer in the new element to NULL to signal that it is the last item in the list.

Here is the code:

```
person *current;
...
current = head;
while (current->next != NULL)
    current = current->next;
new = (person*)malloc(sizeof(struct person));
current->next = new;
new->next = NULL;
```

Figure 15.10 illustrates the procedure for adding a new element to the end of a linked list.

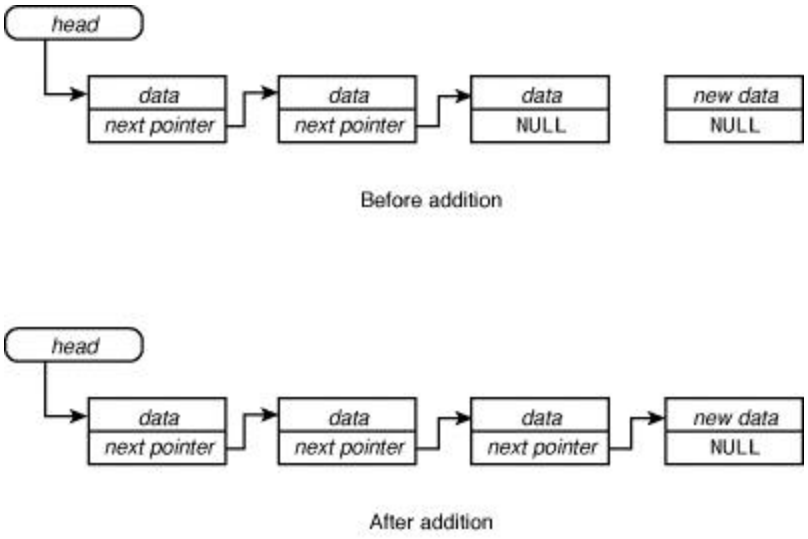

Figure 15-10. Adding a new element to the end of a linked list.

Adding an Element to the Middle of the List

When you're working with a linked list, most of the time you will be adding elements somewhere in the middle of the list. Exactly where the new element is placed depends on how you're keeping the list--for example, if it is sorted on one or more data elements. This process, then, requires that you first locate the position in the list where the new element will go, and then add it. Here are the steps to follow:

1. In the list, locate the existing element that the new element will be placed after. Let's call this the *marker* element.
2. Create an instance of your structure, allocating memory space using malloc().
3. Set the next pointer of the marker element to point to the new element (whose address is returned by malloc()).
4. Set the next pointer of the new element to point to the element that the marker element used to point to.

Here's how the code might look:

```
person *marker;
/* Code here to set marker to point to the desired list location. */
...
new = (LINK)malloc(sizeof(PERSON));
new->next = marker->next;
marker->next = new;
```

Figure 15.11 illustrates this process.

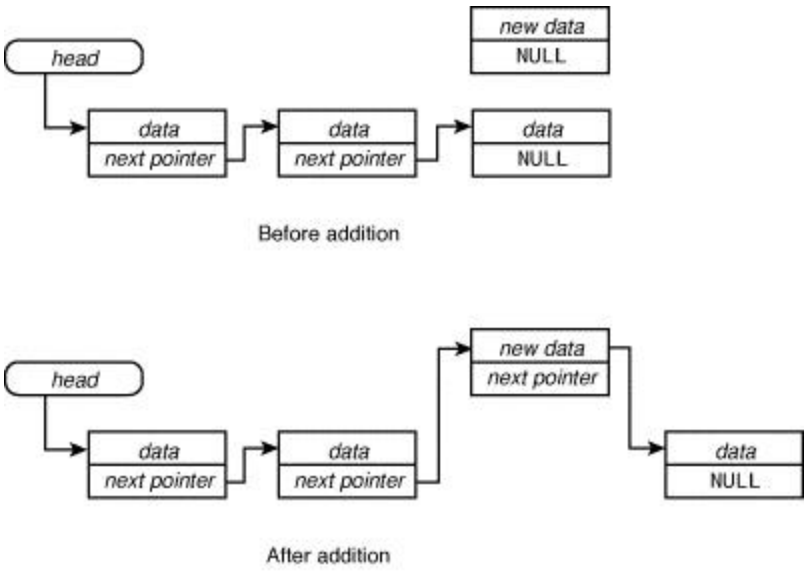

Figure 5-11. Adding a new element to the middle of a linked list.

Deleting an Element from the List

Deleting an element from a linked list is a simple matter of manipulating pointers. The exact process depends on where in the list the element is located:

- To delete the first element, set the head pointer to point to the second element in the list.
- To delete the last element, set the next pointer of the next-to-last element to NULL.
- To delete any other element, set the next pointer of the element before the one being deleted to point to the element after the one being deleted.

Here's the code to delete the first element in a linked list:

```
head = head->next;
```

This code deletes the last element in the list:

```
person *current1, *current2;
current1 = head;
current2= current1->next;
while (current2->next != NULL)
{
    current1 = current2;
    current2= current1->next;
}
current1->next = null;
if (head == current1)
    head = null;
```

Finally, the following code deletes an element from within the list:

```
person *current1, *current2;
/* Code goes here to have current1 point to the */
/* element just before the one to be deleted. */
current2 = current1->next;
current1->next = current2->next;
```

After any of these procedures, the deleted element still exists in memory, but it is removed from the list because there is no pointer in the list pointing to it. In a real-world program, you would want to reclaim the memory occupied by the "deleted" element. This is accomplished with the free() function. You'll learn about this function in detail on Day 20, "Working with Memory."

15-5-3. A Simple Linked List Demonstration

Listing 15.12 demonstrates the basics of using a linked list. This program is clearly for demonstration purposes only, because it doesn't accept user input and doesn't do anything useful other than show the code required for the most basic linked list tasks. The program does the following:

1. It defines a structure and the required pointers for the list.
2. It adds the first element to the list.
3. It adds an element to the end of the list.
4. It adds an element to the middle of the list.
5. It displays the list contents on-screen.

Listing 15.12. The basics of a linked list.

```
1:  /* Demonstrates the fundamentals of using */
2:  /* a linked list. */
3:
4:  #include <stdlib.h>
5:  #include <stdio.h>
6:  #include <string.h>
7:
8:  /* The list data structure. */
9:  struct data {
10:     char name[20];
11:     struct data *next;
12:     };
13:
14: /* Define typedefs for the structure */
15: /* and a pointer to it. */
16: typedef struct data PERSON;
17: typedef PERSON *LINK;
18:
19: main()
20: {
21: /* Head, new, and current element pointers. */
22: LINK head = NULL;
23: LINK new = NULL;
24: LINK current = NULL;
25:
26: /* Add the first list element. We do not */
```

```
27: /* assume the list is empty, although in */
28: /* this demo program it always will be. */
29:
30: new = (LINK)malloc(sizeof(PERSON));
31: new->next = head;
32: head = new;
33: strcpy(new->name, "Abigail");
34:
35: /* Add an element to the end of the list. */
36: /* We assume the list contains at least one element. */
37:
38: current = head;
39: while (current->next != NULL)
40: {
41:     current = current->next;
42: }
43:
44: new = (LINK)malloc(sizeof(PERSON));
45: current->next = new;
46: new->next = NULL;
47: strcpy(new->name, "Catherine");
48:
49: /* Add a new element at the second position in the list. */
50: new = (LINK)malloc(sizeof(PERSON));
51: new->next = head->next;
52: head->next = new;
53: strcpy(new->name, "Beatrice");
54:
55: /* Print all data items in order. */
56: current = head;
57: while (current != NULL)
58: {
59:     printf("\n%s", current->name);
60:     current = current->next;
61: }
62:
63: printf("\n");
64: return(0);
65: }
Abigail
Beatrice
Catherine
```

ANALYSIS: You can probably figure out at least some of the code. Lines 9 through 12 declare the data structure for the list. Lines 16 and 17 define typedefs for both the data structure and for a pointer to the data structure. Strictly speaking, this isn't necessary, but it simplifies coding by allowing you to write PERSON in place of struct data and LINK in place of struct data *.

Lines 22 through 24 declare a head pointer and a couple of other pointers that will be used when manipulating the list. All of these pointers are initialized to NULL.

Lines 30 through 33 add a new link to the start of the list. Line 30 allocates a new data structure. Note that the successful operation of malloc() is assumed--something you would never do in a real program!

Line 31 sets the next pointer in this new structure to point to whatever the head pointer contains. Why not simply assign NULL to this pointer? That works only if you know that the list is empty. As it is written, the code will work even if the list already contains some elements. The new first element will end up pointing to the element that used to be first, which is just what you want.

Line 32 makes the head pointer point to the new record, and line 33 stores some data in the record.

Adding an element to the end of the list is a bit more complicated. Although in this case you know that the list contains only one element, you can't assume this in a real program. Therefore, it's necessary to loop through the list, starting with the first element, until you find the last element (as indicated by the next pointer's being NULL). Then you know you have found the end of the list. This task is accomplished in lines 38 through 42. Once you have found the last element, it is a simple matter to allocate a new data structure, have the old last element point to it, and set the new element's next pointer to NULL, because it is now the last element in the list. This is done in lines 44 through 47.

The next task is to add an element to the middle of the list--in this case, at the second position. After a new data structure is allocated (line 50), the new element's next pointer is set to point to the element that used to be second and is now third in the list (line 51), and the first element's next pointer is made to point to the new element (line 52).

Finally, the program prints all the records in the linked list. This is a simple matter of starting with the element that the head pointer points to and then progressing through the list until the last list element is found, as indicated by a NULL pointer. Lines 56 through 61 perform this task.

15-5-4. Implementing a Linked List

Now that you have seen the ways to add links to a list, it's time to see them in action. Listing 15.13 is a rather long program that uses a linked list to hold a list of five characters. The characters are stored in memory using a linked list. These characters just as easily could have been names, addresses, or any other data. To keep the example as simple as possible, only a single character is stored in each link.

What makes this linked list program complicated is the fact that it sorts the links as they are added. Of course, this also is what makes this program so valuable. Each link is added to the beginning, middle, or end, depending on its value. The link is always sorted. If you were to write a program that simply added the links to the end, the logic would be much simpler. However, the program also would be less useful.

Listing 15.13. Implementing a linked list of characters.

```
1:  /*=========================================================*
2:   * Program:  list1513.c                                    *
3:   * Book:     Teach Yourself C in 21 Days                   *
4:   * Purpose:  Implementing a linked list                    *
5:   *=========================================================*/
6:  #include <stdio.h>
7:  #include <stdlib.h>
8:
9:  #ifndef NULL
10: #define NULL 0
11: #endif
12:
13: /* List data structure */
14: struct list
15: {
16:   int    ch;     /* using an int to hold a char */
17:   struct list *next_rec;
```

```
18: };
19:
20: /* Typedefs for the structure and pointer. */
21: typedef struct list LIST;
22: typedef LIST *LISTPTR;
23:
24: /* Function prototypes. */
25: LISTPTR add_to_list( int, LISTPTR );
26: void show_list(LISTPTR);
27: void free_memory_list(LISTPTR);
28:
29: int main( void )
30: {
31:    LISTPTR first = NULL;  /* head pointer */
32:    int i = 0;
33:    int ch;
34:    char trash[256];       /* to clear stdin buffer. */
35:
36:    while ( i++ < 5 )      /* build a list based on 5 items given
*/
37:    {
38:       ch = 0;
39:       printf("\nEnter character %d, ", i);
40:
41:       do
42:       {
43:           printf("\nMust be a to z: ");
44:           ch = getc(stdin);  /* get next char in buffer  */
45:           gets(trash);       /* remove trash from buffer */
46:       } while( (ch < `a' || ch > `z') && (ch < `A' || ch > `Z'));
47:
48:       first = add_to_list( ch, first );
49:    }
50:
51:    show_list( first );           /* Dumps the entire list */
52:    free_memory_list( first );    /* Release all memory */
53:    return(0);
54: }
55:
56: /*=======================================================*
57:  * Function: add_to_list()
58:  * Purpose : Inserts new link in the list
59:  * Entry   : int ch = character to store
60:  *           LISTPTR first = address of original head pointer
61:  * Returns : Address of head pointer (first)
62:  *=======================================================*/
63:
64: LISTPTR add_to_list( int ch, LISTPTR first )
65: {
66:    LISTPTR new_rec = NULL;       /* Holds address of new rec */
67:    LISTPTR tmp_rec = NULL;       /* Hold tmp pointer         */
```

```
68:    LISTPTR prev_rec = NULL;
69:
70:    /* Allocate memory. */
71:    new_rec = (LISTPTR)malloc(sizeof(LIST));
72:    if (!new_rec)      /* Unable to allocate memory */
73:    {
74:       printf("\nUnable to allocate memory!\n");
75:       exit(1);
76:    }
77:
78:    /* set new link's data */
79:    new_rec->ch = ch;
80:    new_rec->next_rec = NULL;
81:
82:    if (first == NULL)   /* adding first link to list */
83:    {
84:        first = new_rec;
85:        new_rec->next_rec = NULL;  /* redundant but safe */
86:    }
87:    else    /* not first record */
88:    {
89:       /* see if it goes before the first link */
90:       if ( new_rec->ch < first->ch)
91:       {
92:          new_rec->next_rec = first;
93:          first = new_rec;
94:       }
95:       else   /* it is being added to the middle or end */
96:       {
97:          tmp_rec = first->next_rec;
98:          prev_rec = first;
99:
100:         /* Check to see where link is added. */
101:
102:         if ( tmp_rec == NULL )
103:         {
104:             /* we are adding second record to end */
105:             prev_rec->next_rec = new_rec;
106:         }
107:         else
108:         {
109:            /* check to see if adding in middle */
110:            while (( tmp_rec->next_rec != NULL))
111:            {
112:               if( new_rec->ch < tmp_rec->ch )
113:               {
114:                  new_rec->next_rec = tmp_rec;
115:                  if (new_rec->next_rec != prev_rec->next_rec)
116:                  {
117:                     printf("ERROR");
118:                     getc(stdin);
```

```
119:                     exit(0);
120:                  }
121:                  prev_rec->next_rec = new_rec;
122:                  break;   /* link is added; exit while */
123:               }
124:               else
125:               {
126:                 tmp_rec = tmp_rec->next_rec;
127:                 prev_rec = prev_rec->next_rec;
128:               }
129:            }
130:
131:            /* check to see if adding to the end */
132:            if (tmp_rec->next_rec == NULL)
133:            {
134:               if (new_rec->ch < tmp_rec->ch ) /* 1 b4 end */
135:               {
136:                  new_rec->next_rec = tmp_rec;
137:                  prev_rec->next_rec = new_rec;
138:               }
139:               else  /* at the end */
140:               {
141:                  tmp_rec->next_rec = new_rec;
142:                  new_rec->next_rec = NULL;  /* redundant */
143:               }
144:            }
145:         }
146:      }
147:   }
148:   return(first);
149: }
150:
151: /*=========================================================*
152:  * Function: show_list
153:  * Purpose : Displays the information current in the list
154:  *=========================================================*/
155:
156: void show_list( LISTPTR first )
157: {
158:    LISTPTR cur_ptr;
159:    int counter = 1;
160:
161:    printf("\n\nRec addr  Position  Data  Next Rec addr\n");
162:    printf("========  ========  ====  =============\n");
163:
164:    cur_ptr = first;
165:    while (cur_ptr != NULL )
166:    {
167:       printf("  %X   ", cur_ptr );
168:       printf("     %2i       %c", counter++, cur_ptr->ch);
169:       printf("      %X   \n",cur_ptr->next_rec);
```

```
170:       cur_ptr = cur_ptr->next_rec;
171:    }
172: }
173:
174: /*=========================================================*
175:  * Function: free_memory_list
176:  * Purpose : Frees up all the memory collected for list
177:  *=========================================================*/
178:
179: void free_memory_list(LISTPTR first)
180: {
181:    LISTPTR cur_ptr, next_rec;
182:    cur_ptr = first;                 /* Start at beginning */
183:
184:    while (cur_ptr != NULL)          /* Go while not end of list
*/
185:    {
186:       next_rec = cur_ptr->next_rec; /* Get address of next record
*/
187:       free(cur_ptr);                /* Free current record */
188:       cur_ptr = next_rec;           /* Adjust current record*/
189:    }
190: }
Enter character 1,
Must be a to z: q
Enter character 2,
Must be a to z: b
Enter character 3,
Must be a to z: z
Enter character 4,
Must be a to z: c
Enter character 5,
Must be a to z: a
Rec addr  Position  Data  Next Rec addr
========  ========  ====  =============
C3A          1       a      C22
C22          2       b      C32
C32          3       c      C1A
C1A          4       q      C2A
C2A          5       z      0
```

NOTE: Your output will probably show different address values.

ANALYSIS: This program demonstrates adding a link to a linked list. It isn't the easiest listing to understand; however, if you walk through it, you'll see that it's a combination of the three methods of adding links that were discussed earlier. This listing can be used to add links to the beginning, middle, or end of a linked list. Additionally, this listing takes into consideration the special cases of adding the first link (the one that gets added to the beginning) and the second link (the one that gets added to the middle).

TIP: The easiest way to fully understand this listing is to step through it line-by-line in your compiler's debugger and to read the following analysis. By seeing the logic executed, you will better understand the listing.

Several items at the beginning of Listing 15.13 should be familiar or easy to understand. Lines 9 through 11 check to see whether the value of NULL is already defined. If it isn't, line 10 defines it to be 0. Lines 14 through 22 define the structure for the linked list and also declare the type definitions to make working with the structure and pointers easier.

The main() function should be easy to follow. A head pointer called first is declared in line 31. Notice that this is initialized to NULL. Remember that you should never let a pointer go uninitialized. Lines 36 through 49 contain a while loop that is used to get five characters from the user. Within this outer while loop, which repeats five times, a do...while is used to ensure that each character entered is a letter. The isalpha() function could have been used just as easily.

After a piece of data is obtained, add_to_list() is called. The pointer to the beginning of the list and the data being added to the list are passed to the function.

The main() function ends by calling show_list() to display the list's data and then free_memory_list() to release all the memory that was allocated to hold the links in the list. Both these functions operate in a similar manner. Each starts at the beginning of the linked list using the head pointer first. A while loop is used to go from one link to the next using the next_ptr value. When next_ptr is equal to NULL, the end of the linked list has been reached, and the functions return.

The most important (and most complicated!) function in this listing is add_to_list() in lines 56 through 149. Lines 66 through 68 declare three pointers that will be used to point to three different links. The new_rec pointer will point to the new link that is to be added. The tmp_rec pointer will point to the current link in the list being evaluated. If there is more than one link in the list, the prev_rec pointer will be used to point to the previous link that was evaluated.

Line 71 allocates memory for the new link that is being added. The new_rec pointer is set to the value returned by malloc(). If the memory can't be allocated, lines 74 and 75 print an error message and exit the program. If the memory was allocated successfully, the program continues.

Line 79 sets the data in the structure to the data passed to this function. This simply consists of assigning the character passed to the function ch to the new record's character field (new_rec->ch). In a more complex program, this could entail the assigning of several fields. Line 80 sets the next_rec in the new record to NULL so that it doesn't point to some random location.

Line 82 starts the "add a link" logic by checking to see whether there are any links in the list. If the link being added is the first link in the list, as indicated by the head pointer first being NULL, the head pointer is simply set equal to the new pointer, and you're done.

If this link isn't the first, the function continues within the else at line 87. Line 90 checks to see whether the new link goes at the beginning of the list. As you should remember, this is one of the three cases for adding a link. If the link does go first, line 92 sets the next_rec pointer in the new link to point to the previous "first" link. Line 93 then sets the head pointer, first, to point to the new link. This results in the new link's being added to the beginning of the list.

If the new link isn't the first link to be added to an empty list, and if it's being added at the first position in an existing list, you know it must be in the middle or at the end of the list. Lines 97 and 98 set up the tmp_rec and prev_rec pointers that were declared earlier. The pointer tmp_rec is set to the address of the second link in the list, and prev_rec is set to the first link in the list.

You should note that if there is only one link in the list, tmp_rec will be equal to NULL. This is because tmp_rec is set to the next_ptr in the first link, which will be equal to NULL. Line 102 checks for this special case. If tmp_rec is

NULL, you know that this is the second link being added to the list. Because you know the new link doesn't come before the first link, it can only go at the end. To accomplish this, you simply set prev_rec->next_ptr to the new link, and then you're done.

If the tmp_rec pointer isn't NULL, you know that you already have more than two links in your list. The while statement in lines 110 through 129 loops through the rest of the links to determine where the new link should be placed. Line 112 checks to see whether the new link's data value is less than the link currently being pointed to. If it is, you know this is where you want to add the link. If the new data is greater than the current link's data, you need to look at the next link in the list. Lines 126 and 127 set the pointers tmp_rec and next_rec to the next links.

If the character is "less than" the current link's character, you would follow the logic presented earlier in this chapter for adding to the middle of a linked list. This process can be seen in lines 114 through 122. In line 114, the new link's next pointer is set to equal the current link's address (tmp_rec). Line 121 sets the previous link's next pointer to point to the new link. After this, you're done. The code uses a break statement to get out of the while loop.

NOTE: Lines 115 through 120 contain debugging code that was left in the listing for you to see. These lines could be removed; however, as long as the program is running correctly, they will never be called. After the new link's next pointer is set to the current pointer, it should be equal to the previous link's next pointer, which also points to the current record. If they aren't equal, something went wrong!

The previously covered logic takes care of links being added to the middle of the list. If the end of the list is reached, the while loop in lines 110 through 129 will end without adding the link. Lines 132 through 144 take care of adding the link to the end.

If the last link in the list was reached, tmp_rec->next_rec will equal NULL. Line 132 checks for this condition. Line 134 checks to see whether the link goes before or after the last link. If it goes after the last link, the last link's next_rec is set to the new link (line 132), and the new link's next pointer is set to NULL (line 142).

Improving Listing 15.13

Linked lists are not the easiest thing to learn. As you can see from Listing 15.13, however, they are an excellent way of storing data in a sorted order. Because it's easy to add new data items anywhere in a linked list, the code for keeping a list of data items in sorted order with a linked list is a lot simpler that it would be if you used, say, an array. This listing could easily be converted to sort names, phone numbers, or any other data. Additionally, although this listing sorted in ascending order (A to Z), it just as easily could have sorted in descending order (Z to A).

Deleting from a Linked List

The capability to add information to a linked list is essential, but there will be times when you will want to remove information too. Deleting links, or elements, is similar to adding them. You can delete links from the beginning, middle, or end of linked lists. In each case, the appropriate pointers need to be adjusted. Also, the memory used by the deleted link needs to be freed.

NOTE: Don't forget to free memory when deleting links!

DON'T forget to free any memory allocated for links when deleting them.

DO understand the difference between calloc() and malloc(). Most important, remember that malloc() doesn't initialize allocated memory--calloc() does.

15-6. Summary

This chapter covered some of the advanced uses of pointers. As you probably realize by now, pointers are a central part of the C language. C programs that don't use pointers are rare. You saw how to use pointers to pointers and how arrays of pointers can be very useful when dealing with strings. You also learned how C treats multidimensional arrays as arrays of arrays, and you saw how to use pointers with such arrays. You learned how to declare and use pointers to functions, an important and flexible programming tool. Finally, you learned how to implement linked lists, a powerful and flexible data storage method.

This has been a long and involved chapter. Although some of its topics are a bit complicated, they're exciting as well. With this chapter, you're really getting into some of the sophisticated capabilities of the C language. Power and flexibility are among the main reasons C is such a popular language.

Q&A

Q How many levels deep can I go with pointers to pointers?
A You need to check your compiler manuals to determine whether there are any limitations. It's usually impractical to go more than three levels deep with pointers (pointers to pointers to pointers). Most programs rarely go over two levels.
Q Is there a difference between a pointer to a string and a pointer to an array of characters?
A No. A string can be considered an array of characters.
Q Is it necessary to use the concepts presented in this chapter to take advantage of C?
A You can use C without ever using any advanced pointer concepts; however, you won't take advantage of the power that C offers. By using pointer manipulations such as those shown in this chapter, you should be able to perform virtually any programming task in a quick, efficient manner.
Q Are there other times when function pointers are useful?
A Yes. Pointers to functions also are used with menus. Based on a value returned from a menu, a pointer is set to an appropriate function.
Q Name two major advantages of linked lists.
A One: The size of a linked list can be increased or decreased while the program is running, and it doesn't have to be predefined when you write the code. Two: It's easy to keep a linked list in sorted order, because elements can easily be added or deleted anywhere in the list.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned.

Quiz

1. Write code that declares a type float variable, declares and initializes a pointer to the variable, and declares and initializes a pointer to the pointer.
2. Continuing with the example in question 1, say that you want to use the pointer to a pointer to assign the value 100 to the variable x. What, if anything, is wrong with the following assignment statement?

```
*ppx = 100;
```

If it isn't correct, how should it be written?
3. Assume that you have declared an array as follows:

```
int array[2][3][4];
```

What is the structure of this array, as seen by the C compiler?
4. Continuing with the array declared in question 3, what does the expression array[0][0] mean?

5. Again using the array from question 3, which of the following comparisons is true?

```
array[0][0] == &array[0][0][0];
array[0][1] == array[0][0][1];
array[0][1] == &array[0][1][0];
```

6. Write the prototype for a function that takes an array of pointers to type char as its one argument and returns void.
7. How would the function that you wrote a prototype for in question 6 know how many elements are in the array of pointers passed to it?
8. What is a pointer to a function?
9. Write a declaration of a pointer to a function that returns a type char and takes an array of pointers to type char as an argument.
10. You might have answered question 9 with

```
char *ptr(char *x[]);
```

What is wrong with this declaration?
11. When defining a data structure to be used in a linked list, what is the one element that must be included?
12. What does it mean if the head pointer is equal to NULL?
13. How are single-linked lists connected?
14. What do the following declare?
a. int *var1;
b. int var2;
c. int **var3;
15. What do the following declare?
a. int a[3][12];
b. int (*b)[12];
c. int *c[12];
16. What do the following declare?
a. char *z[10];
b. char *y(int field);
c. char (*x)(int field);

Exercises

1. Write a declaration for a pointer to a function that takes an integer as an argument and returns a type float variable.
2. Write a declaration for an array of pointers to functions. The functions should all take a character string as a parameter and return an integer. What could such an array be used for?
3. Write a statement to declare an array of 10 pointers to type char.
4. BUG BUSTER: Is anything wrong with the following code?

```
int x[3][12];
int *ptr[12];
ptr = x;
```

5. Write a structure that is to be used in a single-linked list. This structure should hold your friends' names and addresses.
Because of the many possible solutions, answers are not provided for the following exercises.
6. Write a program that declares a 12*12 array of characters. Place Xs in every other element. Use a pointer to the array to print the values to the screen in a grid format.
7. Write a program that uses pointers to type double variables to accept 10 numbers from the user, sort them, and print them to the screen. (Hint: See Listing 15.10.)
8. Modify the program in exercise 10 to allow the user to specify whether the sort order is ascending or descending.