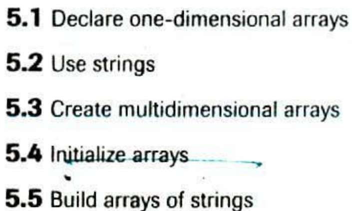
5

Exploring Arrays and Strings

chapter objectives



5.1 Declare one-dimensional arrays

5.2 Use strings

5.3 Create multidimensional arrays

5.4 Initialize arrays

5.5 Build arrays of strings

IN this chapter you will learn about arrays. An *array* is essentially a list of related variables and can be very useful in a variety of situations. Since in C strings are simply arrays of characters, you will also learn about strings and several of C's string functions.

Before proceeding, you should be able to answer these questions and perform these exercises:

1. What is the difference between a local and a global variable?
2. What data type will a C compiler assign to these numbers? (Assume 16-bit integers.)
   a. 10
   b. 10000
   c. 123.45
   d. 123564
   e. −45099
3. Write a program that inputs a **long**, a **short**, and a **double** and then writes these values to the screen.
4. What does a type cast do?
5. To which **if** is the **else** in this fragment associated? What is the general rule?

```
if(i)
  if(j)  printf("i and j are true");
else printf("i is false");
```

6. Using the following fragment, what is the value of **a** when **i** is 1? What is **a**'s value when **i** is 4?

```
switch(i) {
  case 1: a = 1;
  case 2: a = 2;
    break;
  case 3: a = 3;
    break;
```

```
    case 4:
    case 5: a = 5;
  }
```

5.1 DECLARE ONE-DIMENSIONAL ARRAYS

In C, a one-dimensional array is a list of variables that are all of the same type and are accessed through a common name. An individual variable in the array is called an *array element*. Arrays form a convenient way to handle groups of related data.

To declare a one-dimensional array, use the general form

type var_name[size];

where *type* is a valid C data type, *var_name* is the name of the array, and *size* specifies the number of elements in the array. For example, to declare an integer array with 20 elements called **myarray**, use this statement.

```
int myarray[20];
```

An array element is accessed by indexing the array using the number of the element. In C, all arrays begin at zero. This means that if you want to access the first element in an array, use zero for the index. To index an array, specify the index of the element you want inside square brackets. For example, the following refers to the second element of **myarray**:

```
myarray[1]
```

Remember, arrays start at zero, so an index of 1 references the second element.

To assign an array element a value, put the array on the left side of an assignment statement. For example, this gives the first element in **myarray** the value 100:

```
myarray[0] = 100;
```

C stores one-dimensional arrays in one contiguous memory location with the first element at the lowest address. For example, after this fragment executes,

```
int i[5];
int j;

for(j=0; j<5; j++) i[j] = j;
```

array **i** will look like this:

	i[0]	i[1]	i[2]	i[3]	i[4]
i	**0**	**1**	**2**	**3**	**4**

You may use the value of an array element anywhere you would use a simple variable or constant. For example, the following program loads the **sqrs** array with the squares of the numbers 1 through 10 and then displays them.

```
#include <stdio.h>

int main(void)
{
  int sqrs[10];
  int i;

  for(i=1; i<11; i++) sqrs[i-1] = i*i;

  for(i=0; i<10; i++) printf("%d ", sqrs[i]);

  return 0;
}
```

When you want to use **scanf( )** to input a numeric value into an array element, simply put the **&** in front of the array name. For example, this call to **scanf( )** reads an integer into **count[9]**.

```
scanf("%d", &count[9]);
```

C does not perform any bounds checking on array indexes. This means that it is possible to overrun the end of an array. For example, if an array called **a** is declared as having five elements, the compiler will still let you access the (nonexistent) tenth element with a statement like **a[9]**. Of course, attempting to access nonexistent elements will generally have disastrous results, often causing the

program to crash. It is up to you, the programmer, to make sure that the ends of arrays are never overrun.

In C, you may not assign one entire array to another. For example, this fragment is incorrect.

```
char a1[10], a2[10];
.
.
.
a2 = a1; /* this is wrong */
```

If you wish to copy the values of all the elements of one array to another, you must do so by copying each element separately.

EXAMPLES

1. Arrays are very useful when lists of information need to be managed. For example, this program reads the noonday temperature for each day of a month and then reports the month's average temperature, as well as its hottest and coolest days.

```
#include <stdio.h>

int main(void)
{
  int temp[31], i, min, max, avg;
  int days;

  printf("How many days in the month? ");
  scanf("%d", &days);

  for(i=0; i<days; i++) {
    printf("Enter noonday temperature for day %d", i+1);
    scanf("%d", &temp[i]);
  }

  /* find average */
  avg = 0;
  for(i=0; i<days; i++) avg = avg + temp[i];
```

```
  printf("Average temperature: %d\n", avg/days);

  /* find min and max */
  min = 200;  /* initialize min and max */
  max = 0;
  for(i=0; i<days; i++) {
    if(min>temp[i]) min = temp[i];
    if(max<temp[i]) max = temp[i];
  }

  printf("Minimum temperature: %d\n", min);
  printf("Maximum temperature: %d\n", max);

  return 0;
}
```

2. As stated earlier, to copy the contents of one array to another, you must explicitly copy each element separately. For example, this program loads **a1** with the numbers 1 through 10 and then copies them into **a2**.

```
#include <stdio.h>

int main(void)
{
  int a1[10], a2[10];
  int i;

  for(i=1; i<11; i++) a1[i-1] = i;

  for(i=0; i<10; i++) a2[i] = a1[i];

  for(i=0; i<10; i++) printf("%d ", a2[i]);

  return 0;
}
```

3. The following program is an improved version of the code-machine program developed in Chapter 3. In this version, the user first enters the message, which is stored in a character array. When the user presses ENTER, the entire message is then encoded by adding 1 to each letter.

```
#include <stdio.h>
#include <conio.h>

int main(void)
{
  char mess[80];
  int i;

  printf("Enter message (less than 80 characters)\n");
  for(i=0; i<80; i++) {
    mess[i] = getche();
    if(mess[i]=='\r') break;
  }
  printf("\n");

  for(i=0; mess[i]!='\r'; i++) printf("%c", mess[i]+1);

  return 0;
}
```

4. Arrays are especially useful when you want to sort information. For example, this program lets the user enter up to 100 numbers and then sorts them. The sorting algorithm is the bubble sort. The bubble sort algorithm is not very efficient, but it is simple to understand and easy to code. The general concept behind the bubble sort, indeed how it got its name, is the repeated comparisons and, if necessary, exchanges of adjacent elements. This is a little like bubbles in a tank of water with each bubble, in turn, seeking its own level.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  int item[100];
  int a, b, t;
  int count;

  /* read in numbers */
  printf("How many numbers? ");
  scanf("%d", &count);
  for(a=0; a<count; a++) scanf("%d", &item[a]);
```

```
  /* now, sort them using a bubble sort */
  for(a=1; a<count; ++a)
    for(b=count-1; b>=a; --b) {
      /* compare adjacent elements */
      if(item[b-1] > item[b]) {
        /* exchange elements */
        t = item[b-1];
        item[b-1] = item[b];
        item[b] = t;
      }
    }

  /* display sorted list */
  for(t=0; t<count; t++) printf("%d ", item[t]);

  return 0;
}
```

EXERCISES

1. What is wrong with this program fragment?

```
#include <stdio.h>

int main(void)
{
  int i, count[10];

  for(i=0; i<100; i++) {
    printf("Enter a number: ");
    scanf("%d", &count[i]);
  }
  .
  .
  .
```

2. Write a program that reads ten numbers entered by the user and reports if any of them match.
3. Change the sorting program shown in the examples so that it sorts data of type **float**.

USE STRINGS

The most common use of the one-dimensional array in C is the string. Unlike most other computer languages, C has no built-in string data type. Instead, a string is defined as a *null-terminated character array*. In C, a null is zero. The fact that string must be terminated by a null means that you must define the array that is going to hold a string to be one byte larger than the largest string it will be required to hold, to make room for the null. A string constant is null-terminated by the compiler automatically.

There are several ways to read a string from the keyboard. The method we will use in this chapter employs another of C's standard library functions: **gets( )**. Like the other standard I/O functions, **gets( )** also uses the STDIO.H header file. To use **gets( )**, call it using the name of a character array without any index. The **gets( )** function reads characters until you press ENTER. The ENTER key (i.e., carriage return) is not stored, but is replaced by a null, which terminates the string. For example, this program reads a string entered at the keyboard. It then displays the contents of that string one character at a time.

```
#include <stdio.h>

int main(void)
{
  char str[80];
  int i;

  printf("Enter a string (less than 80 chars): ");
  gets(str);
  for(i=0; str[i]; i++) printf("%c", str[i]);

  return 0;
}
```

Notice how the program uses the fact that a null is false to control the loop that outputs the string.

There is a potential problem with **gets( )** that you need to be aware of. The **gets( )** function performs no bounds checking, so it is possible for the user to enter more characters than the array receiving them can hold. For example, if you call **gets( )** with an array that is 20 characters long, there is no mechanism to stop you from entering

more than 20 characters. If you *do* enter more than 20 characters, the array will be overrun. This can obviously lead to trouble, including a program crash. Later in this book you will learn some alternative ways to read strings, although none are as convenient as using **gets( )**. For now, just be sure to call **gets( )** with an array that is more than large enough to hold the expected input.

In the previous program, the string that was entered by the user was output to the screen a character at a time. There is, of course, a much easier way to display a string using **printf( )**, as shown in this version of the program:

```
#include <stdio.h>

int main(void)
{
  char str[80];

  printf("Enter a string (less than 80 chars): ");
  gets(str);
  printf(str); /* output the string */

  return 0;
}
```

Recall that the first argument to **printf( )** is a string. Since **str** contains a string it can be used as the first argument to **printf( )**. The contents of **str** will then be displayed.

If you wanted to output other items in addition to **str**, you could display **str** using the **%s** format code. For example, to output a newline after **str**, you could use this call to **printf( )**.

```
printf("%s\n", str);
```

This method uses the **%s** format specifier followed by the newline character and uses **str** as a second argument to be matched by the **%s** specifier.

The C standard library supplies many string-related functions. The four most important are **strcpy( )**, **strcat( )**, **strcmp( )**, and **strlen( )**. These functions require the header file STRING.H. Let's look at each now.

The **strcpy( )** function has this general form:

strcpy(*to*, *from*);

It copies the contents of *from* to *to*. The contents of *from* are unchanged. For example, this fragment copies the string "hello" into **str** and displays it on the screen:

```
char str[80];

strcpy(str, "hello");
printf("%s", str);
```

The **strcpy( )** function performs no bounds checking, so you must make sure that the array on the receiving end is large enough to hold what is being copied, including the null terminator.

The **strcat( )** function adds the contents of one string to another. This is called *concatenation*. Its general form is

strcat(*to*, *from*);

It adds the contents of *from* to the contents of *to*. It performs no bounds checking, so you must make sure that *to* is large enough to hold its current contents plus what it will be receiving. This fragment displays **hello there**.

```
char str[80];

strcpy(str, "hello");
strcat(str, " there");
printf(str);
```

The **strcmp( )** function compares two strings. It takes this general form:

strcmp(*s1*, *s2*);

It returns zero if the strings are the same. It returns less than zero if *s1* is less than *s2* and greater than zero if *s1* is greater than *s2*. The strings are compared lexicographically; that is, in dictionary order. Therefore, a string is less than another when it would appear before the other in a dictionary. A string is greater than another when it would appear after the other. The comparison is not based upon the length of the string. Also, the comparison is case-sensitive, lowercase characters being greater than uppercase. This fragment prints **0**, because the strings are the same:

```
printf("%d", strcmp("one", "one"));
```

The **strlen( )** function returns the length, in characters, of a string. Its general form is

strlen(*str*);

The **strlen( )** function does not count the null terminator. This means that if **strlen( )** is called using the string "test", it will return 4.

EXAMPLES

1. This program requests input of two strings, then demonstrates the four string functions with them.

```
#include <string.h>
#include <stdio.h>

int main(void)
{
  char str1[80], str2[80];
  int i;

  printf("Enter the first string: ");
  gets(str1);
  printf("Enter the second string: ");
  gets(str2);

  /* see how long the strings are */
  printf("%s is %d chars long\n", str1, strlen(str1));
  printf("%s is %d chars long\n", str2, strlen(str2));

  /* compare the strings */
  i = strcmp(str1, str2);
  if(!i) printf("The strings are equal.\n");
  else if(i<0) printf("%s is less than %s\n", str1, str2);
  else printf("%s is greater than %s\n", str1, str2);

  /* concatenate str2 to end of str1 if
     there is enough room */
  if(strlen(str1) + strlen(str2) < 80) {
    strcat(str1, str2);
    printf("%s\n", str1);
```

```
  /* copy str2 to str1 */
  strcpy(str1, str2);
  printf("%s %s\n", str1, str2);

  return 0;
}
```

One common use of strings is to support a *command-based interface*. Unlike a menu, which allows the user to make a selection, a command-based interface displays a prompting message, waits for the user to enter a command, and then does what the command requests. Many operating systems, such as Windows or DOS, support command-line interfaces, for example. The following program is similar to a program developed in Section 3.1. It allows the user to add, subtract, multiply, or divide, but does not use a menu. Instead, it uses a command-based interface.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
  char command[80], temp[80];
  int i, j;

  for( ; ; ) {
    printf("Operation? ");
    gets(command);

    /* see if user wants to stop */
    if(!strcmp(command, "quit")) break;

    printf("Enter first number: ");
    gets(temp);
    i = atoi(temp);

    printf("Enter second number: ");
    gets(temp);
    j = atoi(temp);
```

```
    /*  now, perform the operation */
    if(!strcmp(command, "add"))
      printf("%d\n", i+j);
    else if(!strcmp(command, "subtract"))
      printf("%d\n", i-j);
    else if(!strcmp(command, "divide")) {
      if(j) printf("%d\n", i/j);
    }
    else if(!strcmp(command, "multiply"))
      printf("%d\n", i*j);
    else printf("Unknown command. \n");
  }

  return 0;
}
```

Notice that this example also introduces another of C's standard library functions: **atoi( )**. The **atoi( )** function returns the integer equivalent of the number represented by its string argument. For example, **atoi("100")** returns the value 100. The reason that **scanf( )** is not used to read the numbers is because, in this context, it is incompatible with **gets( )**. (You will need to know more about C before you can understand the cause of this incompatibility.) The **atoi( )** function uses the header file STDLIB.H.

3. You can create a zero-length string using a **strcpy( )** statement like this:

```
strcpy(str, "");
```

Such a string is called a *null string*. It contains only one element: the null terminator.

EXERCISES

1. Write a program that inputs a string, then displays it backward on the screen.
2. What is wrong with this program?

```
#include <string.h>
#include <stdio.h>

int main(void)
{
  char str[5];

  strcpy(str, "this is a test");
  printf(str);

  return 0;
}
```

3. Write a program that repeatedly inputs strings. Each time a string is input, concatenate it with a second string called **bigstr**. Add newlines to the end of each string. If the user types **quit**, stop inputting and display **bigstr** (which will contain a record of all strings input). Also stop if **bigstr** will be overrun by the next concatenation.

CREATE MULTIDIMENSIONAL ARRAYS

In addition to one-dimensional arrays, you can create arrays of two or more dimensions. For example, to create a 10x12 two-dimensional integer array called **count**, you would use this statement:

```
int count[10][12];
```

As you can see, to add a dimension, you simply specify its size inside square brackets.

A two-dimensional array is essentially an array of one-dimensional arrays and is most easily thought of in a row, column format. For example, given a 4x5 integer array called **two_d**, you can think of it looking like that shown in Figure 5-1. Assuming this conceptual view, a two-dimensional array is accessed a row at a time, from left to right. This means that the rightmost index will change most quickly when the array is accessed sequentially from the lowest to highest memory address.

	0	1	2	3	4
0					
1					
2					
3					

FIGURE 5-2

A conceptual view of a 4x5 two-dimensional array

Two-dimensional arrays are used like one-dimensional ones. For example, this program loads a 4x5 array with the products of the indices, then displays the array in row, column format.

```
#include <stdio.h>

int main(void)
{
  int twod[4][5];
  int i, j;

  for(i=0; i<4; i++)
    for(j=0; j<5; j++)
      twod[i][j] = i*j;

  for(i=0; i<4; i++) {
    for(j=0; j<5; j++)
      printf("%d ", twod[i][j]);
    printf("\n");
  }

  return 0;
}
```

The program output looks like this:

```
0 0 0 0 0
0 1 2 3 4
0 2 4 6 8
0 3 6 9 12
```

To create arrays of three dimensions or greater, simply add the size of the additional dimension. For example, the following statement creates a 10x12x8 three-dimensional array.

```
float values[10][12][8];
```

A three-dimensional array is essentially an array of two-dimensional arrays.

You may create arrays of more than three dimensions, but this is seldom done because the amount of memory they consume increases exponentially with each additional dimension. For example, a 100-character one-dimensional array requires 100 bytes of memory. A 100x100 character array requires 10,000 bytes, and a 100x100x100 array requires 1,000,000 bytes. A 100x100x100x100 four-dimensional array would require 100,000,000 bytes of storage—large even by today's standards.

EXAMPLE

1. A good use of a two-dimensional array is to manage lists of numbers. For example, you could use this two-dimensional array to hold the noontime temperature for each day of the year, grouped by month.

   ```
   float yeartemp[12][31];
   ```

   In the same vein, the following program can be used to keep track of the number of points scored per quarter by each member of a basketball team.

   ```
   #include <stdio.h>

   int main(void)
   {
     int bball[4][5];
     int i, j;

     for(i=0; i<4; i++)
       for(j=0; j<5; j++) {
         printf("Quarter %d, player %d, ", i+1, j+1);
         printf("Enter number of points: ");
   ```

```
            scanf("%d", &bball[i][j]);
        }

    /* display results */
    for(i=0; i<4; i++)
      for(j=0; j<5; j++) {
        printf("Quarter %d, player %d, ", i+1, j+1);
        printf("%d\n", bball[i][j]);
      }

    return 0;
}
```

EXERCISES

1. Write a program that defines a 3x3x3 three-dimensional array, and load it with the numbers 1 to 27.
2. Have the program from the first exercise display the sum of its elements.

INITIALIZE ARRAYS

Like other types of variables, you can give the elements of arrays initial values. This is accomplished by specifying a list of values the array elements will have. The general form of array initialization for one-dimensional arrays is shown here:

type array-name[*size*] = {*value-list* };

The *value-list* is a comma-separated list of constants that are type compatible with the base type of the array. Moving from left to right, the first constant will be placed in the first position of the array, the second constant in the second position, and so on. Note that a semicolon follows the }. In the following example, a five-element integer array is initialized with the squares of the numbers 1 through 5.

```
int i[5] = {1, 4, 9, 16, 25};
```

This means that **i[0]** will have the value 1 and **i[4]** will have the value 25.

You can initialize character arrays two ways. First, if the array is not holding a null-terminated string, you simply specify each character using a comma-separated list. For example, this initializes **a** with the letters 'A', 'B', and 'C'.

```
char a[3] = {'A', 'B', 'C'};
```

If the character array is going to hold a string, you can initialize the array using a quoted string, as shown here:

```
char name[5] = "Herb";
```

Notice that no curly braces surround the string. They are not used in this form of initialization. Because strings in C must end with a null, you must make sure that the array you declare is long enough to include the null. This is why **name** is 5 characters long, even though "Herb" is only 4. When a string constant is used, the compiler automatically supplies the null terminator.

Multidimensional arrays are initialized in the same way as one-dimensional arrays. For example, here the array **sqr** is initialized with the values 1 through 9, using row order:

```
int sqr[3][3] = {
  1, 2, 3,
  4, 5, 6,
  7, 8, 9
};
```

This initialization causes **sqr[0][0]** to have the value 1, **sqr[0][1]** to contain 2, **sqr[0][2]** to hold 3, and so forth.

If you are initializing a one-dimensional array, you need not specify the size of the array—simply put nothing inside the square brackets. If you don't specify the size, the compiler counts the number of initializers and uses that value as the size of the array. For example,

```
int pwr[] = {1, 2, 4, 8, 16, 32, 64, 128};
```

causes the compiler to create an initialized array eight elements long. Arrays that don't have their dimensions explicitly specified are called *unsized arrays*. An unsized array is useful because the size of the array

will be automatically adjusted when you change the number of its initializers. It also helps avoid counting errors on long lists, which is especially important when initializing strings. For example, here an unsized array is used to hold a prompting message.

```
char prompt[] = "Enter your name: ";
```

If, at a later date, you wanted to change the prompt to "Enter your last name:", you would not have to count the characters and then change the array size. The size of **prompt** would automatically be adjusted.

Unsized array initializations are not restricted to one-dimensional arrays. However, for multidimensional arrays you must specify all but the leftmost dimension to allow C to index the array properly. In this way you may build tables of varying lengths with the compiler allocating enough storage for them automatically. For example, the declaration of **sqr** as an unsized array is shown here:

```
int sqr[][3] = {
  1, 2, 3,
  4, 5, 6,
  7, 8, 9
};
```

The advantage to this declaration over the sized version is that tables may be lengthened or shortened without changing the array dimensions.

EXAMPLES

1. A common use of an initialized array is to create a lookup table. For example, in this program a 5x2 two-dimensional array is initialized so that the first element in each row is the number of a file server in a network and the second element contains the number of users connected to that server. The program allows a user to enter the number of a server. It then looks up the server in the table and reports the number of users.

```
#include <stdio.h>

int main(void)
{
```

```
  int ServerUsers[5][2] = {
    1, 14,
    2, 28,
    3, 19,
    4, 8,
    5, 15
  };

  int server;
  int i;

  printf("Enter the server number: ");
  scanf("%d", &server);

  /* look it up in the table */
  for(i=0; i<5; i++)
    if(server == ServerUsers[i][0]) {
      printf("There are %d users on server %d.\n",
             ServerUsers[i][1], server);
      break;
    }

  /* report error if not found */
  if(i==5) printf("Server not listed.\n");

  return 0;
}
```

2. Even though an array has been given an initial value, its contents may be changed. For example, this program prints **hello** on the screen.

```
#include <stdio.h>
#include <string.h>

int main(void)
{
  char str[80] = "I like C";

  strcpy(str, "hello");
  printf(str);

  return 0;
}
```

As this program illustrates, in no way does an initialization fix the contents of an array.

EXERCISES

1. Is this fragment correct?

```
int balance[] = 10.0, 122.23, 100.0;
```

2. Is this fragment correct?

```
#include <stdio.h>
#include <string.h>

int main(void)
{
  char name[] = "Tom";

  strcpy(name, "Tom Brazwell");
```

3. Write a program that initializes a 10x3 array so that the first element of each row contains a number, the second element contains its square, and the third element contains its cube. Start with 1 and stop at 10. For example, the first few rows will look like this:

   1, 1, 1,
   2, 4, 8,
   3, 9, 27,
   4, 16, 64,
   .
   .
   .

   Next, prompt the user for a cube, look up this value in the table, and report the cube's root and the root's square. Use an unsized array so that the table size may be easily changed.

5.5 BUILD ARRAYS OF STRINGS

Arrays of strings, often called *string tables*, are very common in C programming. A string table is created like any other two-dimensional array. However, the way you think about it will be slightly different. For example, here is a small string table. What do you think it defines?

```
char names[10][40];
```

This statement specifies a table that can contain 10 strings, each up to 40 characters long (including the null terminator). To access a string within this table, specify only the left-most index. For example, to read a string from the keyboard into the third string in **names**, use this statement:

```
gets(names[2]);
```

By the same token, to output the first string, use this **printf( )** statement:

```
printf(names[0]);
```

The declaration that follows creates a three-dimensional table with three lists of strings. Each list is five strings long, and each string can hold 80 characters.

```
char animals[3][5][80];
```

To access a specific string in this situation, you must specify the two left-most indexes. For example, to access the second string in the third list, specify **animals[2][1]**.

EXAMPLES

1. This program lets you enter ten strings, then lets you display them, one at a time, in any order you choose. To stop the program, enter a negative number.

```
#include <stdio.h>

int main(void)
```

```
int main(void)
  char text[10][80];
  int i;

  for(i=0; i<10; i++) {
    printf("%d: ", i+1);
    gets(text[i]);
  }

  do {
    printf("Enter number of string (1-10) : ");
    scanf("%d", &i);
    i--;  /* adjust value to match array index */
    if(i>=0 && i<10) printf("%s\n", text[i]);
  } while(i>=0);

  return 0;
}
```

program lets you enter ten strings, then lets you display them, one a time in any order you chose, to stop the program, enter a negative number.

2. You can initialize a string table as you would any other type of array. For example, the following program uses an initialized string table to translate between German and English. Notice that curly braces are needed to surround the list. The only time they are not needed is when a single string is being initialized.

```
/* English-to-German Translator. */

#include <stdio.h>
#include <string.h>

char words[][2][40] = {
  "dog", "Hund",
  "no", "nein",
  "year", "Jahr",
  "child", "Kind",
  "I",  "Ich",
  "drive", "fahren",
  "house", "Haus",
  "to", "zu",
  "",""
};

int main(void)
{
  char english[80];
```

```
  int i;

  printf("Enter English word: ");
  gets(english);

  /* look up the word */
  i = 0;
  /* search while null string not yet encountered */
  while(strcmp(words[i][0], "")) {
    if(!strcmp(english, words[i][0])) {
      printf("German translation: %s", words[i][1]);
      break;
    }
    i++;
  }
  if(!strcmp(words[i][0], ""))
    printf("Not in dictionary\n");

  return 0;
}
```

3. You can access the individual characters that comprise a string within a string table by using the rightmost index. For example, the following program prints the strings in the table one character at a time.

```
#include <stdio.h>

int main(void)
{
  char text[][80] = {
    "When", "in", "the",
    "course", "of", "human",
    "events", ""
  };

  int i, j;

  /* now, display them */
  for(i=0; text[i][0]; i++) {
    for(j=0; text[i][j]; j++)
      printf("%c", text[i][j]);
    printf(" ");
  }
  return 0;
}
```

```
    return 0;
}
```

EXERCISE

1. Write a program that creates a string table containing the English words for the numbers 0 through 9. Using this table, allow the user to enter a digit (as a character) and then have your program display the word equivalent. (Hint: to obtain an index into the table, subtract '0' from the character entered.)

Mastery Skills Check

At this point you should be able to perform these exercises and answer these questions:

1. What is an array?
2. Given the array

   ```
   int count[10];
   ```

   will this statement generate an error message?

   ```
   for(i=0; i<20; i++) count[i] = i;
   ```

3. In statistics, the *mode* of a group of numbers is the one that occurs the most often. For example, given the list 1, 2, 3, 6, 4, 7, 5, 4, 6, 9, 4, the mode is 4, because it occurs three times. Write a program that allows the user to enter a list of 20 numbers and then finds and displays the mode.
4. Show how to initialize an integer array called **items** with the values 1 through 10.
5. Write a program that repeatedly reads strings from the keyboard until the user enters **quit**.
6. Write a program that acts like an electronic dictionary. If the user enters a word in the dictionary, the program displays its

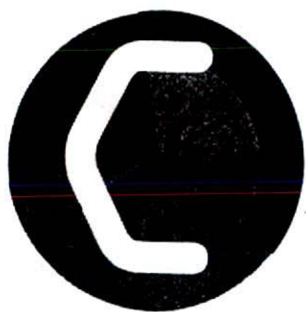
meaning. Use a three-dimensional character array to hold the words and their meanings.

This section checks how well you have integrated the material in this chapter with that from earlier chapters.

1. Write a program that inputs strings from the user. If the string is less than 80 characters long, pad it with periods. Print out the string to verify that you have correctly lengthened the string.
2. Write a program that inputs a string and then encodes it by taking the characters from each end, starting with the left side and alternating, stopping when the middle of the string has been reached. For example, the string "Hi there" would be "Heir eth".
3. Write a program that counts the number of spaces, commas, and periods in a string. Use a **switch** to categorize the characters.
4. What is wrong with this fragment?

```
char str[80];
str = getchar();
```

5. Write a program that plays a computerized version of Hangman. In the game of Hangman, you are shown the length of a magic word (using hyphens) and you try to guess what the word is by entering letters. Each time you enter a letter, the magic word is checked to see if it contains that letter. If it does, that letter is shown. Keep a count on the number of letters entered to complete the word. For the sake of simplicity, a player wins when the magic word is entirely filled by characters using 15 or fewer guesses. For this exercise make the magic word "concatenation."



6

Using Pointers

chapter objectives

6.1 Understand pointer basics

6.2 Learn restrictions to pointer expressions

6.3 Use pointers with arrays

6.4 Use pointers to string constants

6.5 Create arrays of pointers

6.6 Become acquainted with multiple indirection

6.7 Use pointers as parameters

THIS chapter covers one of C's most important and sometimes most troublesome features: the *pointer*. A pointer is basically the address of an object. One reason that pointers are so important is that much of the power of the C language is derived from the unique way in which they are implemented. You will learn about the special pointer operators, pointer arithmetic, and how arrays and pointers are related. Also, you will be introduced to using pointers as parameters to functions.

Before proceeding, you should be able to answer these questions and perform these exercises:

1. Write a program that inputs 10 integers into an array. Then have the program display the sum of the even numbers and the sum of the odd numbers.
2. Write a program that simulates a log-on to a remote system. The system can be accessed only if the user knows the password, which in this case is "Tristan." Give the user three tries to enter the correct password. If the user succeeds, simply print **Log-on Successful** and exit. If the user fails after three attempts to enter the correct password, display **Access Denied** and exit.
3. What is wrong with this fragment?

```
char name[10] = "Thomas Jefferson";
```

4. What is a null string?
5. What does **strcpy( )** do? What does **strcmp( )** do?
6. Write a program that creates a string table consisting of names and telephone numbers. Initialize the array with some names of people you know and their phone numbers. Next, have the program request a name and print the associated telephone number. In other words, create a computerized telephone book.

UNDERSTAND POINTER BASICS

A pointer is a variable that holds the memory address of another object. For example, if a variable called **p** contains the address of another variable called **q**, then **p** is said to *point* to **q**. Therefore if **q** is at location 100 in memory, then **p** would have the value 100.

To declare a pointer variable, use this general form:

*type *var-name;*

Here, *type* is the *base type* of the pointer. The base type specifies the type of the object that the pointer can point to. Notice that the variable name is preceded by an asterisk. This tells the computer that a pointer variable is being created. For example, the following statement creates a pointer to an integer:

```
int *p;
```

C contains two special pointer operators: * and &. The & operator returns the address of the variable it precedes. The * operator returns the value stored at the address that it precedes. (The * pointer operator has no relationship to the multiplication operator, which uses the same symbol.) For example, examine this short program:

```
#include <stdio.h>

int main(void)
{
  int *p, q;

  q = 199; /* assign q 199 */

  p = &q; /* assign p the address of q */

  printf("%d", *p); /* display q's value using pointer */

  return 0;
}
```

This program prints **199** on the screen. Let's see why.

First, the line

```
int *p, q;
```

defines two variables: **p**, which is declared as an integer pointer, and **q**, which is an integer. Next, **q** is assigned the value 199. In the next line, **p** is assigned the *address of* **q**. You can verbalize the & operator as "address of." Therefore, this line can be read as "assign **p** the address of **q**." Finally, the value is displayed using the * operator applied to **p**. The * operator can be verbalized as "at address." Therefore, the **printf( )** statement can be read as "print the value at address **q**," which is 199.

When a variable's value is referenced through a pointer, the process is called *indirection*.

It is possible to use the * operator on the left side of an assignment statement in order to assign a variable a new value given a pointer to it. For example, this program assigns **q** a value indirectly using the pointer **p**:

```
#include <stdio.h>

int main(void)
{
  int *p, q;

  p = &q; /* get q's address */

  *p = 199; /* assign q a value using a pointer */

  printf("q's value is %d", q);

  return 0;
}
```

In the two simple example programs just shown, there is no reason to use a pointer. However, as you learn more about C, you will understand why pointers are important. Pointers are used to support linked lists and binary trees, for example.

The base type of a pointer is very important. Although C allows any type of pointer to point anywhere in memory, it is the base type that determines how the object pointed to will be treated. To understand the importance of this, consider the following fragment:

```
int q;
double *fp;

fp = &q;

/* what does this line do? */
*fp = 100.23;
```

Although not syntactically incorrect, this fragment is wrong. The pointer **fp** is assigned the address of an integer. This address is then used on the left side of an assignment statement to assign a floating-point value. However, **ints** are usually shorter than **doubles**, and this assignment statement causes memory adjacent to **q** to be overwritten. For example, in an environment in which integers are 2 bytes and **doubles** are 8 bytes, the assignment statement uses the 2 bytes allocated to **q** as well as 6 adjacent bytes, thus causing an error.

In general, the C compiler uses the base type to determine how many bytes are in the object pointed to by the pointer. This is how C knows how many bytes to copy when an indirect assignment is made, or how many bytes to compare if an indirect comparison is made. Therefore, it is very important that you always use the proper base type for a pointer. Except in special cases, never use a pointer of one type to point to an object of a different type.

If you attempt to use a pointer before it has been assigned the address of a variable, your program will probably crash. Remember, declaring a pointer variable simply creates a variable capable of holding a memory address. It does not give it any meaningful initial value. This is why the following fragment is incorrect.

```
int main(void)
{
  int *p;

  *p = 10;   /* incorrect - p is not pointing to
                anything */
```

As the comment notes, the pointer **p** is not pointing to any known object. Hence, trying to indirectly assign a value using **p** is meaningless and dangerous.

As pointers are defined in C, a pointer that contains a null value (zero) is assumed to be unused and pointing at nothing. In C, a null is, by convention, assumed to be an invalid memory address. However,

the compiler will still let you use a null pointer, usually with disastrous results.

Examples

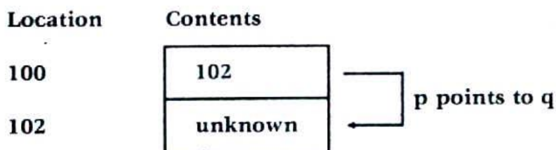
1. To graphically illustrate how indirection works, assume these declarations:

```
int *p, q;
```

Further assume that **q** is located at memory address 102 and that **p** is right before it, at location 100. After this statement

```
p = &q;
```

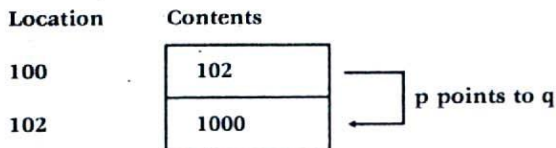
the pointer **p** contains the value 102. Therefore, after this assignment, memory looks like this:



After the statement

```
*p = 1000;
```

executes, memory looks like this:



Remember, the value of **p** has nothing to do with the *value* of **q**. It simply holds **q**'s *address*, to which the indirection operator may be applied.

2. To illustrate why you must make sure that the base type of a pointer is the same as the object it points to, try this incorrect but benign program. (Some compilers may generate a warning message when you compile it, but none will issue an actual error message and stop compilation.)

```
/* This program is wrong, but harmless. */

#include <stdio.h>

int main(void)
{
  int *p;
  double q, temp;

  temp = 1234.34;

  p = &temp; /* attempt to assign q a value using */
  q = *p;    /* indirection through an integer pointer */

  printf("%f", q); /* this will not print 1234.34 */

  return 0;
}
```

Even though **p** points to **temp**, which does, indeed, hold the value 1234.34, the assignment

```
q = *p;
```

fails to copy the number because only 2 bytes (assuming 2-byte integers) will be transferred. Since **p** is an integer pointer, it cannot be used to transfer an 8-byte quantity (assuming 8-byte **doubles**).

EXERCISES

1. What is a pointer?
2. What are the pointer operators and what are their effects?

3. Why is the base type of a pointer important?
4. Write a program with a **for** loop that counts from 0 to 9, displaying the numbers on the screen. Print the numbers using a pointer.

LEARN RESTRICTIONS TO POINTER EXPRESSIONS

In general, pointers may be used like other variables. However, you need to understand a few rules and restrictions.

In addition to the * and & operators, there are only four other operators that may be applied to pointer variables: the arithmetic operators +, ++, -, and --. Further, you may add or subtract only integer quantities. You cannot, for example, add a floating-point number to a pointer.

Pointer arithmetic differs from "normal" arithmetic in one very important way: it is performed relative to the base type of the pointer. Each time a pointer is incremented, it will point to the next item, as defined by its base type, beyond the one currently pointed to. For example, assume that an integer pointer called **p** contains the address 200. After the statement

```
p++;
```

executes, **p** will have the value 202, assuming integers are two bytes long. By the same token, if **p** had been a **float** pointer (assuming 4-byte **floats**), then the resultant value contained in **p** would have been 204.

The only pointer arithmetic that appears as "normal" occurs when **char** pointers are used. Because characters are one byte long, an increment increases the pointer's value by one, and a decrement decreases its value by one.

You may add or subtract any integer quantity to or from a pointer. For example, the following is a valid fragment:

```
int *p
.
.
.
p = p + 200;
```

This statement causes **p** to point to the 200th integer past the one to which **p** was previously pointing.

Aside from addition and subtraction of an integer, you may not perform any other type of arithmetic operations—you may not multiply, divide, or take the modulus of a pointer. However, you may subtract one pointer from another in order to find the number of elements separating them.

It is possible to apply the increment and decrement operators to either the pointer itself or the object to which it points. However, you must be careful when attempting to modify the object pointed to by a pointer. For example, assume that **p** points to an integer that contains the value 1. What do you think the following statement will do?

```
*p++;
```

Contrary to what you might think, this statement first increments **p** and then obtains the value at the new location. To increment what is pointed to by a pointer, you must use a form like this:

```
(*p)++;
```

The parentheses cause the value pointed to by **p** to be incremented.

You may compare two pointers using the relational operators. However, pointer comparisons make sense only if the pointers relate to each other—if they both point to the same object, for example. (Soon you will see an example of pointer comparisons.) You may also compare a pointer to zero to see if it is a null pointer.

At this point you might be wondering what use there is for pointer arithmetic. You will shortly see, however, that it is one of the most valuable components of the C language.

EXAMPLES

1. You can use **printf( )** to display the memory address contained in a pointer by using the **%p** format specifier. We can use this **printf( )** capability to illustrate several aspects of pointer arithmetic. The following program, for example, shows how all pointer arithmetic is relative to the base type of the pointer.

```
#include <stdio.h>

int main(void)
{
  char *cp, ch;
  int  *ip, i;
  float  *fp, f;
  double *dp, d;

  cp = &ch;
  ip = &i;
  fp = &f;
  dp = &d;

  /* print the current values */
  printf("%p %p %p %p\n", cp, ip, fp, dp);

  /* now increment them by one */
  cp++;
  ip++;
  fp++;
  dp++;

  /* print their new values */
  printf("%p %p %p %p\n", cp, ip, fp, dp);

  return 0;
}
```

Although the values contained in the pointer variables in this program will vary widely between compilers and even between versions of the same compiler, you will see that the address pointed to by **ch** will be incremented by one byte. The others will be incremented by the number of bytes in their base types. For example, in a 16-bit environment this will typically be 2 for **ints**, 4 for **floats**, and 8 for **doubles**.

2. The following program illustrates the need for parentheses when you want to increment the object pointed to by a pointer instead of the pointer itself.

```
#include <stdio.h>

int main(void)
{
  int *p, q;

  p = &q;

  q = 1;
  printf("%p ", p);

  *p++; /* this will not increment q */
  printf("%d %p", q, p);

  return 0;
}
```

After this program has executed, **q** still has the value 1, but **p** has been incremented. However, if the program is written like this:

```
#include <stdio.h>

int main(void)
{
  int *p, q;

  p = &q;

  q = 1;
  printf("%p ", p);

  (*p)++; /* now q is incremented and p is unchanged */
  printf("%d %p", q, p);

  return 0;
}
```

q is incremented to 2 and **p** is unchanged.

EXERCISES

1. What is wrong with this fragment?

```
int *p, i;

p = &i;

p = p * 8;
```

2. Can you add a floating-point number to a pointer?
3. Assume that **p** is a **float** pointer that currently points to location 100 and that **floats** are 4 bytes long. What is the value of **p** after this fragment has executed?

```
p = p + 2;
```

6.3 USE POINTERS WITH ARRAYS

In C, pointers and arrays are closely related. In fact, they are often interchangeable. It is this relationship between the two that makes their implementation both unique and powerful.

When you use an array name without an index, you are generating a pointer to the start of the array. This is why no indexes are used when you read a string using **gets( )**, for example. What is being passed to **gets( )** is not an array, but a pointer. In fact, you cannot pass an array to a function in C; you may only pass a pointer to the array. This important point was not mentioned in the preceding chapter on arrays because you had not yet learned about pointers. However, this fact is crucial to understanding the C language. The **gets( )** function uses the pointer to load the array it points to with the characters you enter at the keyboard. You will see how this is done later.

Since an array name without an index is a pointer to the start of the array, it stands to reason that you can assign that value to another pointer and access the array using pointer arithmetic. And, in fact, this is exactly what you can do. Consider this program:

```
#include <stdio.h>

int main(void)
{
  int a[10] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
  int *p;

  p = a; /* assign p the address of start of a */

  /* this prints a's first, second and third elements */
  printf("%d %d %d\n", *p, *(p+1), *(p+2));

  /* this does the same thing using a */
  printf("%d %d %d", a[0], a[1], a[2]);

  return 0;
}
```

Here, both **printf( )** statements display the same thing. The parentheses in expressions such as ***(p + 2)** are necessary because the * has a higher precedence than the + operator.

Now you should be able to fully understand why pointer arithmetic is done relative to the base type—it allows arrays and pointers to relate to each other.

To use a pointer to access multidimensional arrays, you must manually do what the compiler does automatically. For example, in this array:

```
float balance[10][5];
```

each row is five elements long. Therefore, to access **balance[3][1]** using a pointer you must use a fragment like this:

```
float *p;

p = (float *) balance;
*(p + (3*5) + 1)
```

To reach the desired element, you must multiply the row number by the number of elements in the row and then add the number of the element within the row. Generally, with multidimensional arrays it is easier to use array indexing rather than pointer arithmetic.

In the preceding example, the cast of **balance** to **float** * was necessary. Since the array is being indexed manually, the pointer arithmetic must be relative to a **float** pointer. However, the type of pointer generated by **balance** is to a two-dimensional array of **floats** Thus, there is need for the cast.

Pointers and arrays are linked by more than the fact that by using pointer arithmetic you can access array elements. You might be surprised to learn that you can index a pointer as if it were an array. The following program, for example, is perfectly valid:

```
#include <stdio.h>

int main(void)
{
  char str[] = "Pointers are fun";
  char *p;
  int i;

  p = str;

  /* loop until null is found */
  for(i=0; p[i]; i++)
    printf("%c", p[i]);

  return 0;
}
```

Keep one point firmly in mind: you should index a pointer only when that pointer points to an array. While the following fragment is syntactically correct, it is wrong; if you tried to execute it, you would probably crash your computer.

```
char *p, ch;
int i;

p = &ch;
for(i=0; i<10; i++) p[i] = 'A'+i;  /* wrong */
```

Since **ch** is not an array, it cannot be meaningfully indexed.

Although you can index a pointer as if it were an array, you will seldom want to do this because pointer arithmetic is usually more convenient. Also, in some cases a C compiler can generate faster

executable code for an expression involving pointers than for a comparable expression using arrays.

Because an array name without an index is a pointer to the start of the array, you can, if you choose, use pointer arithmetic rather than array indexing to access elements of the array. For example, this program is perfectly valid and prints **c** on the screen:

```
#include <stdio.h>

int main(void)
{
  char str[80];

  *(str+3) = 'c';
  printf("%c", *(str+3));

  return 0;
}
```

You cannot, however, modify the value of the pointer generated by using an array name. For example, assuming the previous program, this is an invalid statement:

```
str++;
```

The pointer that is generated by **str** must be thought of as a constant that always points to the start of the array. Therefore, it is invalid to modify it and the compiler will report an error.

EXAMPLES

1. Two of C's library functions, **toupper( )** and **tolower( )**, are called using a character argument. In the case of **toupper( )**, if the character is a lowercase letter, the uppercase equivalent is returned; otherwise the character is returned unchanged. For **tolower( )**, if the character is an uppercase letter, the lowercase equivalent is returned; otherwise the character is returned unchanged. These functions use the header file CTYPE.H. The following program requests a string from the

user and then prints the string, first in uppercase letters and then in lowercase. This version uses array indexing to access the characters in the string so they can be converted into the appropriate case.

```
#include <ctype.h>
#include <stdio.h>

int main(void)
{

  char str[80];
  int i;

  printf("Enter a string: ");
  gets(str);

  for(i=0; str[i]; i++)
    str[i] = toupper(str[i]);

  printf("%s\n", str); /* uppercase string */

  for(i=0; str[i]; i++)
    str[i] = tolower(str[i]);

  printf("%s\n", str); /* lowercase string */

  return 0;
}
```

The same program is shown below, only this time, a pointer is used to access the string. This second approach is the way you would see this program written by professional C programmers because incrementing a pointer is often faster than indexing an array.

```
#include <ctype.h>
#include <stdio.h>

int main(void)
{
  char str[80], *p;

  printf("Enter a string: ");
```

```
  gets(str);
  p = str;

  while(*p) {
    *p = toupper(*p);
    p++;
  }

  printf("%s\n", str); /* uppercase string */

  p = str; /* reset p */

  while(*p) {
    *p = tolower(*p);
    p++;
  }

  printf("%s\n", str); /* lowercase string */

  return 0;
}
```

Before leaving this example, a small digression is in order. The routine

```
while(*p) {
  *p = toupper(*p);
  p++;
}
```

will generally be written by experienced programmers like this:

```
while(*p)
  *p++ = toupper(*p);
```

Because the **++** follows the **p**, the value pointed to by **p** is first modified and then **p** is incremented to point to the next element. Since this is the way C code is often written, this book will use the more compact form from time to time when it seems appropriate.

Remember that although most of the examples have been incrementing pointers, you can decrement a pointer as well. For example, the following program uses a pointer to copy the contents of one string into another in reversed order.

```
#include <stdio.h>
#include <string.h>

int main(void)
{
  char str1[] = "Pointers are fun to use";
  char str2[80], *p1, *p2;

  /* make p point to end of str1 */
  p1 = str1 + strlen(str1) - 1;

  p2 = str2;

  while(p1 >= str1)
    *p2++ = *p1--;

  /* null terminate str2 */
  *p2 = '\0';

  printf("%s %s", str1, str2);

  return 0;
}
```

This program works by setting **p1** to point to the end of **str1**, and **p2** to the start of **str2**. It then copies the contents of **str1** into **str2** in reverse order. Notice the pointer comparison in the **while** loop. It is used to stop the copying process when the start of **str1** is reached.

Also, notice the use of the compacted forms ***p2++** and ***p1--**. The loop is the equivalent of this one:

```
while(p1 >= str1) {
  *p2 = *p1;
  p1--;
  p2++;
}
```

Again, it is important for you to become familiar with the compact form of these types of pointer operations.

EXERCISES

1. Is this fragment correct?

```
int count[10];
.
.
.
count = count + 2;
```

2. What value does this fragment display?

```
int temp[5] = {10, 19, 23, 8, 9};
int *p;

p = temp;

printf("%d", *(p+3));
```

3. Write a program that inputs a string. Have the program look for the first space. If it finds one, print the remainder of the string.

USE POINTERS TO STRING CONSTANTS

As you know, C allows string constants enclosed between double quotes to be used in a program. When the compiler encounters such a string, it stores it in the program's string table and generates a pointer to the string. For this reason, the following program is correct and prints **one two three** on the screen.

```
#include <stdio.h>

int main(void)
{
  char *p;

  p = "one two three";

  printf(p);

  return 0;
}
```

Let's see how this program works. First, **p** is declared as a character pointer. This means that it may point to an array of characters. When the compiler compiles the line

```
p = "one two three";
```

it stores the string in the program's string table and assigns to **p** the address of the string in the table. Therefore, when **p** is used in the **printf( )** statement, **one two three** is displayed on the screen.

This program can be written more efficiently, as shown here:

```
#include <stdio.h>

int main(void)
{
  char *p = "one two three";

  printf(p);

  return 0;
}
```

Here, **p** is initialized to point to the string.

EXAMPLES

1. This program continues to read strings until you enter **stop**:

```
#include <stdio.h>
#include <string.h>

int main(void)
{
  char *p = "stop";
  char str[80];

  do {
   printf("Enter a string: ");
   gets(str);
  } while(strcmp(p, str));
```

```
  return 0;
}
```

2. Using pointers to string constants can be very helpful when those constants are quite long. For example, suppose that you had a program that at various times would prompt the user to insert a diskette into drive A. To save yourself some typing, you might elect to initialize a pointer to the string and then simply use the pointer when the message needed to be displayed; for example:

```
char *InsDisk = "Insert disk into drive A, then press ENTER";
.
.
.
printf(InsDisk);
.
.
.
printf(InsDisk);
```

Another advantage to this approach is that to change the prompt, you only need to change it once, and all references to it will reflect the change.

EXERCISE

1. Write a program that creates three character pointers and initialize them so that one points to the string "one", the second to the string "two", and the third to the string "three". Next, have the program print all six permutations of these three strings. (For example, one permutation is "one two three", another is "two one three".)

CREATE ARRAYS OF POINTERS

Pointers may be arrayed like any other data type. For example, the following statement declares an integer pointer array that has 20 elements:

```
int *pa[20];
```

The address of an integer variable called **myvar** is assigned to the ninth element of the array as follows:

```
pa[8] = &myvar;
```

Because **pa** is an array of pointers, the only values that the array elements may hold are the addresses of integer variables. To assign the integer pointed to by the third element of **pa** the value 100, use the statement:

```
*pa[2] = 100;
```

EXAMPLES

1. Probably the single most common use of arrays of pointers is to create string tables in much the same way that unsized arrays were used in the previous chapter. For example, this function displays an error message based on the value of its parameter **err_num**.

```
char *p[] = {
  "Input exceeds field width",
  "Out of range",
  "Printer not turned on",
  "Paper out",
  "Disk full",
  "Disk write error"
};

void error(int err_num)
{
  printf(p[err_num]);
}
```

2. The following program uses a two-dimensional array of pointers to create a string table that links apple varieties with their colors. To use the program, enter the name of the apple, and the program will tell you its color.

```
#include <stdio.h>
#include <string.h>

char *p[][2] = {
  "Red Delicious", "red",
  "Golden Delicious", "yellow",
  "Winesap", "red",
  "Gala", "reddish orange",
  "Lodi", "green",
  "Mutsu", "yellow",
  "Cortland", "red",
  "Jonathan", "red",
  "", "" /* terminate the table with null strings */
};

int main(void)
{
  int i;
  char apple[80];

  printf("Enter name of apple: ");
  gets(apple);

  for(i=0; *p[i][0]; i++) {
    if(!strcmp(apple, p[i][0]))
      printf("%s is %s\n", apple, p[i][1]);
  }

  return 0;
}
```

Look carefully at the condition controlling the **for** loop. The expression ***p[i][0]** gets the value of the first byte of the **i**th string. Since the list is terminated by null strings, this value will be zero (false) when the end of the table is reached. In all other cases it will be nonzero, and the loop will repeat.

EXERCISE

1. In this exercise, you will create an "executive decision aid." This is a program that answers yes, no, or maybe to a question entered at the keyboard. To create this program use an array of character pointers and initialize them to point to these three strings: "Yes", "No", and "Maybe. Rephrase the question". Next, input the user's question and find the length of the string. Next, use this formula to compute an index into the pointer array:

   index = length % 3

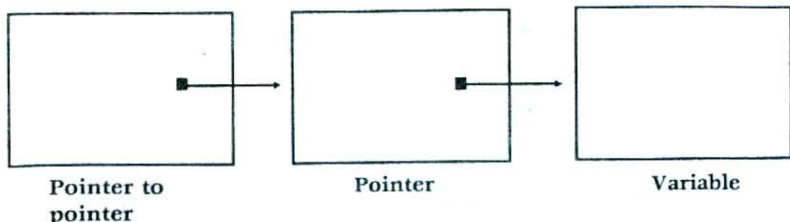
BECOME ACQUAINTED WITH MULTIPLE INDIRECTION

It is possible in C to have a pointer point to another pointer. This is called *multiple indirection* (see Figure 6-1). When a pointer points to another pointer, the first pointer contains the address of the second pointer, which points to the location containing the object.

To declare a pointer to a pointer, an additional asterisk is placed in front of the pointer's name. For example, this declaration tells the compiler that **mp** is a pointer to a character pointer:

```
char **mp;
```

It is important to understand that **mp** is not a pointer to a character, but rather a pointer to a character pointer.



Accessing the target value indirectly pointed to by a pointer to a pointer requires that the asterisk operator be applied twice. For example,

```
char **mp, *p, ch;

p = &ch; /* get address of ch */
mp = &p; /* get address of p */
**mp = 'A'; /* assign ch the value A using multiple
               indirection */
```

As the comments suggest, **ch** is assigned a value indirectly using two pointers.

Multiple indirection is not limited to merely "a pointer to a pointer." You can apply the * as often as needed. However, multiple indirection beyond a pointer to a pointer is very difficult to follow and is not recommended.

You may not see the need for multiple indirection at this time, but as you learn more about C, you will see some examples in which it is very valuable.

EXAMPLES

1. The following program assigns **val** a value using multiple indirection. It displays the value first directly, then through the use of multiple indirection.

```
#include <stdio.h>

int main(void)
{
  float *fp, **mfp, val;

  fp = &val;
  mfp = &fp;

  **mfp = 123.903;
  printf("%f %f", val, **mfp);

  return 0;
```

2. This program shows how you can input a string using **gets( )** by using a pointer to a pointer to the string.

```
#include <stdio.h>

int main(void)
{
  char *p, **mp, str[80];

  p = str;
  mp = &p;

  printf("Enter your name: ");
  gets(*mp);
  printf("Hi %s", *mp);

  return 0;
}
```

Notice that when **mp** is used as an argument to both **gets( )** and **printf( )**, only one * is used. This is because both of these functions require a pointer to a string for their operation. Remember, ****mp** is a pointer to **p**. However, **p** is a pointer to the string **str**. Therefore, ***mp** is a pointer to **str**. If you are a little confused, don't worry. Over time, you will develop a clearer concept of pointers to pointers.

EXERCISE

1. To help you understand multiple indirection better, write a program that assigns an integer a value using a pointer to a pointer. Before the program ends, display the addresses of the integer variable, the pointer, and the pointer to the pointer. (Remember, use **%p** to display a pointer value.)

USE POINTERS AS PARAMETERS

Pointers may be passed to functions. For example, when you call a function like **strlen( )** with the name of a string, you are actually passing a pointer to a function. When you pass a pointer to a function, the function must be declared as receiving a pointer of the same type. In the case of **strlen( )**, this is a character pointer. A complete discussion of using pointers as parameters is presented in the next chapter. However, some basic concepts are discussed here.

When you pass a pointer to a function, the code inside that function has access to the variable pointed to by the parameter. This means that the function can change the variable used to call the function. This is why functions like **strcpy( )**, for example, can work. Because it is passed a pointer, the function is able to modify the array that receives the string.

Now you can understand why you need to precede a variable's name with an *&* when using **scanf( )**. In order for **scanf( )** to modify the value of one of its arguments, it must be passed a pointer to that argument.

EXAMPLES

1. Another of C's standard library functions is called **puts( )**; it writes its string argument to the screen followed by a newline. The program that follows creates its own version of **puts( )** called **myputs( )**.

```
#include <stdio.h>

void myputs(char *p);

int main(void)
{
  myputs("this is a test");
```

```
  return 0;
}

void myputs(char *p)
{
  while(*p) { /* loop as long as p does not point to the
                 null that terminates the string */
    printf("%c", *p);
    p++;        /* go to next character */
  }
  printf("\n");
}
```

This program illustrates a very important point that was mentioned earlier in this chapter. When the compiler encounters a string constant, it places it into the program's string table and generates a pointer to it. Therefore, the **myputs( )** function is actually called with a character pointer, and the parameter **p** must be declared as a character pointer in order to receive it.

2. The following program shows one way to implement the **strcpy( )** function, called **mystrcpy( )**.

```
#include <stdio.h>

void mystrcpy(char *to, char *from);

int main(void)
{
  char str[80];

  mystrcpy(str, "this is a test");
  printf(str);

  return 0;
}

void mystrcpy(char *to, char *from)
{
  while(*from) *to++ = *from++;
    *to = '\0'; /* null terminates the string */
}
```

1. Write your own version of **strcat( )** called **mystrcat( )**, and write a short program that demonstrates it.
2. Write a program that passes a pointer to an integer variable to a function. Inside that function, assign the variable the value -1. After the function has returned, demonstrate that the variable does, indeed, contain -1 by printing its value.

At this point you should be able to perform these exercises and answer these questions:

1. Show how to declare a pointer to a **double**.
2. Write a program that assigns a value to a variable indirectly by using a pointer to that variable.
3. Is this fragment correct? If not, why not?

```
int main(void)
{
  char *p;

  printf("Enter a string: ");
  gets(p);

  return 0;
}
```

4. How do pointers and arrays relate to each other?
5. Given this fragment:

```
char *p, str[80] = "this is a test";

p = str;
```

show two ways to access the 'i' in "this."

6. Assume that **p** is declared as a pointer to a **double** and contains the address 100. Further, assume that **doubles** are 8 bytes long. After **p** is incremented, what will its value be?

Cumulative Skills Check

This section checks how well you have integrated the material in this chapter with that from earlier chapters.

1. What is the advantage of using pointers over array indexing?
2. Below is a program that counts the number of spaces in a string entered by the user. Rewrite the program so that it uses pointer arithmetic rather than array indexing.

```
#include <stdio.h>

int main(void)
{
  char str[80];
  int i, spaces;

  printf("Enter a string: ");
  gets(str);

  spaces = 0;
  for(i=0; str[i]; i++)
    if(str[i]==' ') spaces++;

  printf("Number of spaces: %d", spaces);

  return 0;
}
```

3. Rewrite the following array reference using pointer arithmetic.

```
int count[100][10];

count[44][8] = 99;
```

7

A Closer Look at Functions

chapter objectives

7.1 Understand function prototypes

7.2 Understand recursion

7.3 Take a closer look at parameters

7.4 Pass arguments to **main( )**

7.5 Compare old-style to modern function parameter declarations

AT the very foundation of C is the function. All action statements must appear within one and an understanding of its operation is crucial to successful C programming. This chapter takes a close look at several important topics related to functions.

Review Skills Check

Before proceeding you should be able to answer these questions and perform these exercises:

1. What does this fragment do?

```
int i, *p;

p = &i;
*p = 19;
```

2. What is generated when you use an array name without an index?
3. Is this fragment correct? If it is correct, explain why it works.

```
char *p = "this is a string";
```

4. Write a short program that assigns a floating-point value to a variable indirectly using a pointer to the variable.
5. Write your own version of **strlen( )**, called **mystrlen( )**, and demonstrate it in a program.
6. Is this fragment correct? If it is, what does the program display?

```
char str[8];

strcpy(str, "ABCDEFG");
printf("%c", *(str+2));
```

7.1 UNDERSTAND FUNCTION PROTOTYPES

In Chapter 1 you were briefly introduced to the function prototype. Now it is time for you to understand precisely what a prototype does and why it is important to C programming. Function prototypes were

not supported by the original version of C. They were added when C was standardized in 1989. Many consider prototypes to be the single most important addition made to the C language since its creation. Prototypes are not technically necessary. However, for reasons that will become self-evident, they should be used in all programs that you write.

The general form of a function prototype is shown here:

type function-name(type parameter-name1,
type parameter-name2,
.
.
.
type parameter-nameN);

A prototype declares three attributes associated with a function:

1. Its return type.
2. The number of its parameters.
3. The type of its parameters.

Prototypes provide several benefits. They inform the compiler about the return type of a function. They allow the compiler to find and report illegal type conversions between the type of arguments used to call a function and the type definition of its parameters. Prototypes also enable the compiler to report when the number of arguments passed to a function is not the same as the number of parameters declared by the function. Let's look at each of these.

When you call a function, the compiler needs to know the type of data returned by that function so that it can generate the proper code to handle that data. The reason for this is easy to understand: different data types have different sizes. The code that handles an integer return type will be different from that which handles a **double**, for example. If you use a function that is not prototyped, then the compiler will simply assume that it is returning an integer. However, if it is actually returning some other type, an error will occur. If the function is in the same file as the rest of your program, then the compiler will catch this error. But if the function is in another file or a library, then the error will go uncaught—and this will lead to trouble when your program is executed.

In the absence of a function prototype, it is not syntactically wrong to call a function with incompatible arguments or with more or less

arguments than the function has parameters. Of course, doing either of these is obviously incorrect even though the compiler may accept your program without complaint. The use of a function prototype prevents these errors by enabling the compiler to find them. It is important to understand, however, that not all kinds of type conversions are illegal in a function call. In fact, C automatically converts most types of arguments into the type of data specified by the parameter. But a few type conversions are inherently wrong. For example, you cannot convert an integer into a pointer. A function prototype allows the compiler to catch and return this type of error.

As mentioned, as important as prototypes are, they are not currently required. Because of the need to maintain compatibility with older code, all C compilers still support non-prototyped programs. Of course, at some point in the future, this situation may change.

In early versions of C, before prototypes were invented, it was still necessary to tell the compiler about the return type of a function (unless it returned type **int**) for the reasons explained earlier. This was done using a forerunner of the prototype, called a *forward declaration* or a *forward reference*. A forward declaration is essentially a truncated form of a prototype that declares only the return type of a function—not the type and number of its parameters. Although forward declarations are obsolete, they are still allowed for compatibility with older code.

The following program demonstrates an old-style forward declaration. It uses it to inform the compiler of **volume( )**'s return type.

```
#include <stdio.h>

double volume(); /* old-style forward declaration for
                    volume() */

int main(void)
{
  double vol;

  vol = volume(12.2, 5.67, 9.03);
  printf("Volume: %f", vol);

  return 0;
}

/* Compute the volume of a cube. */
```

```
double volume(double s1, double s2, double s3)
{
  return s1 * s2 * s3;
}
```

Since the old-style declaration does not inform the compiler about any of **volume( )**'s parameters it is not a function prototype. Instead, it simply states **volume( )**'s return type. The trouble is that the lack of a full prototype will allow **volume( )** to be called using an incorrect type and/or number of arguments. For example, given the preceding program, the following will not generate a compiler error message even though it is wrong.

```
volume(120.2, 99.3); /* missing last arg */
```

Since the compiler has not been given information about **volume( )**'s parameters it won't catch the fact that this call is wrong.

Although the old-style forward declaration is no longer used in new code, you will still find it quite frequently in older programs. If you will be updating older programs, you should consider adding prototypes to be your first job.

When function prototypes were added to C, two minor compatibility problems between the old version of C and the ANSI version of C had to be resolved. The first issue was how to handle the old-style forward declaration, which does not use a parameter list. To do so, the ANSI C standard specifies that when a function declaration occurs without a parameter list, nothing whatsoever is being said about the parameters to the function. It might have parameters, it might not. This allows old-style declarations to coexist with prototypes. But it also leads to a question: how do you prototype a function that takes no arguments? For example, this function simply outputs a line of periods:

```
void line()
{
  int i;

  for(i=0; i<80; i++) printf(".");
}
```

If you try to use the following as a prototype, it won't work because the compiler will think that you are simply using the old-style declaration method.

```
void line();
```

The solution to this problem is through the use of the **void** keyword. When a function has no parameters, its prototype uses **void** inside the parentheses. For example, here is **line( )**'s proper prototype:

```
void line(void);
```

This explicitly tells the compiler that the function has no parameters, and any call to that function that has parameters is an error. You must make sure to also use **void** when the function is defined. For example, **line( )** must look like this:

```
void line(void)
{
  int i;

  for(i=0; i<80; i++) printf(".");
}
```

Since we have been using **void** to specify empty parameter lists since Chapter 1, this mechanism is already familiar to you.

The second issue related to prototyping is the way it affects C's automatic type promotions. Because of some features of the environment in which C was developed, when a non-prototyped function is called, all integral promotions take place (for example, characters are converted to integers) and all **floats** are converted to **doubles**. However, these type promotions seem to violate the purpose of the prototype. The resolution to this problem is that when a prototype exists, the types specified in the prototype are maintained, and no type promotions will occur.

There is one other special case that relates to prototypes: variable length argument lists. We won't be creating any functions in this book that use a variable number of arguments because they require the use of some advanced techniques. But it is possible to do so, and it is sometimes quite useful. For example, both **printf( )** and **scanf( )** accept a variable number of arguments. To specify a variable number of arguments, use **...** in the prototype. For example,

```
int myfunc(int a, ...);
```

specifies a function that has one integer parameter and a variable number of other parameters.

In C programming there has been a long-standing confusion about the usage of two terms: *declaration* and *definition*. A declaration specifies the type of an object. A definition causes storage for an object to be created. As these terms relate to functions, a prototype is a *declaration*. The function, itself, which contains the body of the function is a *definition*.

In C, it is also legal to fully define a function prior to its first use, thus eliminating the need for a separate prototype. However, this works only in very small programs. In real-world applications, this option is not feasible. For all practical purposes, function prototypes must exist for all functions that your program will use.

Remember that if a function does not return a value, then its return type should be specified as **void**—both in its definition and in its prototype.

Function prototypes enable you to write better, more reliable programs because they help ensure that the functions in your programs are being called with correct types and numbers of arguments. Fully prototyped programs are the norm and represent the current state of the art of C programming. Frankly, no professional C programmer today would write programs without them. Also, future versions of the ANSI C standard may mandate function prototypes and C++ requires them now. Although prototypes are still technically optional, their use is nearly universal. You should use them in all of the programs you write.

EXAMPLES

1. To see how a function prototype can catch an error, try compiling this version of the volume program, which includes **volume( )**'s full prototype:

```
#include <stdio.h>

/* this is volume()'s full prototype */
double volume(double s1, double s2, double s3);

int main(void)
{
```

```
  double vol;

  vol = volume(12.2, 5.67, 9.03, 10.2); /* error */
  printf("Volume: %f", vol);

  return 0;
}

/* Compute the volume of a cube. */
double volume(double s1, double s2, double s3)
{
  return s1 * s2 * s3;
}
```

As you will see, this program will not compile because the compiler knows that **volume( )** is declared as having only three parameters, but the program is attempting to call it with four parameters.

2. As explained, if a function is defined before it is called, it does not require a separate prototype. For example, the following program is perfectly valid:

```
#include <stdio.h>

/* define getnum() prior to its first use */
float getnum(void)
{
  float x;

  printf("Enter a number: ");
  scanf("%f", &x);
  return x;
}

int main(void)
{
  float i;

  i = getnum();
  printf("%f", i);

  return 0;
}
```

Since **getnum( )** is defined before it is used, the compiler knows what type of data it returns and that it has no parameters. A separate prototype is not needed. The reason that you will seldom use this method is that large programs are typically spread across several files. Since you can't define a function more than once, prototypes are the only way to inform all files about a function. (Multi-file programs are explained in Chapter 11.)

3. As you know, the standard library function **sqrt( )** returns a **double** value. You might be wondering how the compiler knows this. The answer is that **sqrt( )** is prototyped in its header file MATH.H. To see the importance of using the header file, try this program:

```
#include <stdio.h>
/* math.h is intentionally not included */

int main(void)
{
  double answer;

  answer = sqrt(9.0);
  printf("%f", answer);

  return 0;
}
```

When you run this program, it displays something other than **3** because the compiler generates code that copies only two bytes (assuming two-byte integers) into **answer** and not the 8 bytes that typically comprise a **double**. If you include MATH.H, the program will work correctly.

In general, each of C's standard library functions has its prototype specified in a header file. For example, **printf( )** and **scanf( )** have their prototypes in STDIO.H. This is one of the reasons that it is important to include the appropriate header file for each library function you use.

4. There is one situation that you will encounter quite frequently that is, at first, unsettling. Some "character-based" functions have a return type of **int** rather than **char**. For example, the

getchar( ) function's return type is **int**, not **char**. The reason for this is found in the fact that C very cleanly handles the conversion of characters to integers and integers back to characters. There is no loss of information. For example, the following program is perfectly valid:

```
#include <stdio.h>

int get_a_char(void);

int main(void)
{
  char ch;

  ch = get_a_char();
  printf("%c", ch);

  return 0;
}

int get_a_char(void)
{
  return 'a';
}
```

When **get_a_char( )** returns, it elevates the character 'a' to an integer by adding a high-order byte (or bytes) containing zeros. When this value is assigned to **ch** in **main( )**, the high-order byte (or bytes) is removed. One reason to declare functions like **get_a_char( )** as returning an integer instead of a character is to allow various error values to be returned that are intentionally outside the range of a **char**.

5. When a function returns a pointer, both the function and its prototype must declare the same pointer return type. For example, consider this short program:

```
#include <stdio.h>

int *init(int x);
int count;

int main(void)
{
```

```
  int *p;

  p = init(110); /* return pointer */

  printf("count (through p) is %d", *p);

  return 0;
}

int *init(int x)
{
  count = x;

  return &count; /* return a pointer */
}
```

As you can see, the function **init( )** returns a pointer to the global variable **count**. Notice the way that the return type for **init( )** is specified. This same general form is used for any sort of pointer return type. Although this example is trivial, functions that return pointers are quite valuable in many programming situations. One other thing: if a function returns a pointer, then it must make sure that the object being pointed to does not go out-of-scope when the function returns. This means that you must not return pointers to local variables.

6. The **main( )** function does not have (nor does it require) a prototype. This allows you to define **main( )** any way that is supported by your compiler. This book uses

```
int main(void) { ...
```

because it is one of the most common forms. Another frequently used form of **main( )** is shown here:

```
void main(void) { ...
```

This form is used when no value is returned by **main( )**. Later in this chapter, you will see another form of **main( )** that has parameters.

The reason **main( )** does not have a prototype is to allow C to be used in the widest variety of environments. Since the precise conditions present at program start-up and what actions must occur at program termination may differ widely from one

operating system to the next, C allows the acceptable forms of **main( )** to be determined by the compiler. However, nearly all compilers will accept **int main(void)** and **void main(void)**.

EXERCISES

1. Write a program that creates a function, called **avg( )**, that reads ten floating-point numbers entered by the user and returns their average. Use an old-style forward reference and not a function prototype.
2. Rewrite the program from Exercise 1 so that it uses a function prototype.
3. Is the following program correct? If not, why not? If it is, can it be made better?

```
#include <stdio.h>

double myfunc();

int main(void)
{
  printf("%f", myfunc(10.2));

  return 0;
}

double myfunc(double num)
{
  return num / 2.0;
}
```

4. Show the prototype for a function called **Purge( )** that has no parameters and returns a pointer to a **double**.
5. On your own, experiment with the concepts presented in this section.

7.2 UNDERSTAND RECURSION

Recursion is the process by which something is defined in terms of itself. When applied to computer languages, recursion means that a function can call itself. Not all computer languages support recursive functions, but C does. A very simple example of recursion is shown in this program:

```
#include <stdio.h>

void recurse(int i);

int main(void)
{
  recurse(0);

  return 0;
}

void recurse(int i)
{
  if(i<10) {
    recurse(i+1); /* recursive call */
    printf("%d ", i);
  }
}
```

This program prints

```
9 8 7 6 5 4 3 2 1 0
```

on the screen. Let's see why.

The **recurse( )** function is first called with 0. This is **recurse( )**'s first activation. Since 0 is less than 10, **recurse( )** then calls itself with the value of **i** (in this case 0) plus 1. This is the second activation of **recurse( )**, and **i** equals 1. This causes **recurse( )** to be called again using the value 2. This process repeats until **recurse( )** is called with the value 10. This causes **recurse( )** to return. Since it returns to the point of its call, it will execute the **printf( )** statement in its previous activation, print **9**, and return. This, then, returns to the point of its call in the previous activation, which causes **8** to be displayed. The process continues until all the calls return, and the program terminates.

It is important to understand that there are not multiple copies of a recursive function. Instead, only one copy exists. When a function is called, storage for its parameters and local data are allocated on the stack. Thus, when a function is called recursively, the function begins executing with a new set of parameters and local variables, but the code that constitutes the function remains the same.

If you think about the preceding program, you will see that recursion is essentially a new type of program control mechanism. This is why every recursive function you write will have a conditional statement that controls whether the function will call itself again or return. Without such a statement, a recursive function will simply run wild, using up all the memory allocated to the stack and then crashing the program.

Recursion is generally employed sparingly. However, it can be quite useful in simplifying certain algorithms. For example, the Quicksort sorting algorithm is difficult to implement without the use of recursion. If you are new to programming in general, you might find yourself uncomfortable with recursion. Don't worry; as you become more experienced, the use of recursive functions will become more natural.

EXAMPLES

1. The recursive program described above can be altered to print the numbers **0** through **9** on the screen. To accomplish this, only the position of the **printf( )** statement needs to be changed, as shown here:

```
#include <stdio.h>

void recurse(int i);

int main(void)
{
  recurse(0);

  return 0;
}

void recurse(int i)
```

```
{
  if(i<10) {
    printf("%d ", i);
    recurse(i+1);
  }
}
```

Because the call to **printf( )** now precedes the recursive call to **recurse( )**, the numbers are printed in ascending order.

2. The following program demonstrates how recursion can be used to copy one string to another.

```
#include <stdio.h>

void rcopy(char *s1, char *s2);

int main(void)
{
  char str[80];

  rcopy(str, "this is a test");
  printf(str);

  return 0;
}

/* Copy s2 to s1 using recursion. */
void rcopy(char *s1, char *s2)
{
  if(*s2) { /* if not at end of s2 */
    *s1++ = *s2++;
    rcopy(s1, s2);
  }
  else *s1 = '\0'; /* null terminate the string */
}
```

The program works by assigning the character currently pointed to by **s2** to the one pointed to by **s1**, and then incrementing both pointers. These pointers are then used in a recursive call to **rcopy( )**, until **s2** points to the null that terminates the string.

Although this program makes an interesting example of recursion, no professional C programmer would actually code a function like this for one simple reason: efficiency. It takes more time to execute a function call than it does to execute a

loop. Therefore, tasks like this will almost always be coded using an iterative approach.

3. It is possible to have a program in which two or more functions are *mutually recursive*. Mutual recursion occurs when one function calls another, which in turn calls the first. For example, study this short program:

```
#include <stdio.h>

void f2(int b);
void f1(int a);

int main(void)
{
  f1(30);

  return 0;
}

void f1(int a)
{
  if(a) f2(a-1);
  printf("%d ", a);
}

void f2(int b)
{
  printf(".");
  if(b) f1(b-1);
}
```

This program displays

...............0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30

on the screen. Its output is caused by the way the two functions **f1( )** and **f2( )** call each other. Each time **f1( )** is called, it checks to see if **a** is zero. If not, it calls **f2( )** with **a-1**. The **f2( )** function first prints a period and then checks to see if **b** is zero. If not, it calls **f1( )** with **b-1**, and the process repeats. Eventually, **b** is zero and the function calls start unraveling, causing **f1( )** to display the numbers **0** to **30** counting by twos.

EXERCISES

1. One of the best known examples of recursion is the recursive version of a function that computes the factorial of a number. The factorial of a number is obtained by multiplying the original number by all integers less than it and greater than 1. Therefore, 4 factorial is 4x3x2, or 24. Write a function, called **fact( )**, that uses recursion to compute the factorial of its integer argument. Have it return the result. Also, demonstrate its use in a program.

2. What is wrong with this recursive function?

```
void f(void)
{
  int i;

  printf("in f() \n");

  /* call f() 10 times */
  for(i=0; i<10; i++) f( );
}
```

3. Write a program that displays a string on the screen, one character at a time, using a recursive function.

7.3 TAKE A CLOSER LOOK AT PARAMETERS

For computer languages in general, a subroutine can be passed arguments in one of two ways. The first is called *call by value*. This method copies the *value* of an argument into the formal parameter of the subroutine. Therefore, changes made to a parameter of the subroutine *have no effect* on the argument used to call it. The second way a subroutine can have arguments passed to it is through *call by reference*. In this method, the *address* of an argument is copied into the parameter. Inside the subroutine, the address is used to access the actual argument. This means that changes made to the parameter *will affect* the argument.

By default, C uses call by value to pass arguments. This means that you cannot alter the arguments used in a call to a function. What

occurs to a parameter inside the function will have no effect on the argument outside the function. However, as you saw in Chapter 6 , it is possible to manually construct a call by reference by passing a pointer to an argument. Since this causes the address of the argument to be passed, it then is possible to change the value of the argument outside the function.

The classic example of a call-by-reference function is **swap( )**, shown here. It exchanges the value of its two integer arguments.

```
#include <stdio.h>

void swap(int *i, int *j);

int main(void)
{
  int num1, num2;

  num1 = 100;
  num2 = 800;

  printf("num1: %d num2: %d\n", num1, num2);
  swap(&num1, &num2);
  printf("num1: %d num2: %d\n", num1, num2);

  return 0;
}

/* Exchange the values pointed to by two integer pointers. */
void swap(int *i, int *j)
{
  int temp;

  temp = *i;
  *i = *j;
  *j = temp;
}
```

Since pointers to the two integers are passed to the function, the actual values pointed to by the arguments are exchanged.

As you know, when an array is used as an argument to a function, only the address of the array is passed, not a copy of the entire array, which implies call-by-reference. This means that the parameter declaration must be of a compatible pointer type. There are three

ways to declare a parameter that is to receive a pointer to an array. First, the parameter may be declared as an array of the same type and size as that used to call the function. Second, it may be specified as an unsized array. Finally, and most commonly, it may be specified as a pointer to the base type of the array. The following program demonstrates all three methods:

```
#include <stdio.h>

void f1(int num[5]), f2(int num[]), f3(int *num);

int main(void)
{
  int count[5] = {1, 2, 3, 4, 5};

  f1(count);
  f2(count);
  f3(count);

  return 0;
}

/* parameter specified as array */
void f1(int num[5])
{
  int i;

  for(i=0; i<5; i++) printf("%d ", num[i]);
}

/* parameter specified as unsized array */
void f2(int num[])
{
  int i;

  for(i=0; i<5; i++) printf("%d ", num[i]);
}

/* parameter specified as pointer */
void f3(int *num)
{
  int i;
```

```
  for(i=0; i<5; i++) printf("%d ", num[i]);
}
```

Even though the three methods of declaring a parameter that will receive a pointer to an array look different, they all result in a pointer parameter being created.

EXAMPLE

1. Some computer languages, such as BASIC, provide an input function that allows you to specify a prompting message. C has no counterpart for this type of function. However, you can easily create one. The program shown here uses the function **prompt( )** to display a prompting message and then to read a number entered by the user.

```
#include <stdio.h>

void prompt(char *msg, int *num);

int main(void)
{
  int i;
  prompt("Enter a num: ", &i);
  printf("Your number is: %d", i);

  return 0;
}

void prompt(char *msg, int *num)
{
  printf(msg);
  scanf("%d", num);
}
```

Because the parameter **num** is already a pointer, you do not need to precede it with an & in the call to **scanf( )**. (In fact, it would be an error to do so.)

EXERCISES

1. Is this program correct? If not, why not?

```
#include <stdio.h>

myfunc(int num, int min, int max);

int main(void)
{
   int i;

  printf("Enter a number between 1 and 10: ");
  myfunc(&i, 1, 10);

  return 0;
}

void myfunc(int num, int min, int max)
{
  do {
    scanf("%d", num);
  } while(*num<min || *num>max);
}
```

2. Write a program that creates an input function similar to **prompt( )** described earlier in this section. Have it input a string rather than an integer.
3. Explain the difference between call by value and call by reference.

7.4 PASS ARGUMENTS TO main( )

Many programs allow command-line arguments to be specified when they are run. A *command-line argument* is the information that follows the program's name on the command line of the operating system. Command-line arguments are used to pass information into a program. For example, when you use a text editor, you probably specify the name of the file you want to edit after the name of the text editor.

Assuming you use a text editor called EDTEXT, then this line causes the file TEST to be edited.

```
EDTEXT TEST
```

Here, TEST is a command-line argument.

Your C programs may also utilize command-line arguments. These are passed to a C program through two arguments to the **main( )** function. The parameters are called **argc** and **argv**. As you probably guessed, these parameters are optional and are not present when no command-line arguments are being used. Let's look at **argc** and **argv** more closely.

The **argc** parameter holds the number of arguments on the command-line and is an integer. It will always be at least 1 because the name of the program qualifies as the first argument.

The **argv** parameter is an array of string pointers. The most common method for declaring **argv** is shown here:

```
char *argv[];
```

The empty brackets indicate that it is an array of undetermined length. All command-line arguments are passed to **main( )** as strings. To access an individual string, index **argv**. For example, **argv[0]** points to the program's name and **argv[1]** points to the first argument. This program displays all the command-line arguments that are present when it is executed.

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  int i;

  for(i=1; i<argc; i++) printf("%s ", argv[i]);

  return 0;
}
```

C does not specify what constitutes a command-line argument, because operating systems vary considerably on this point. However, the most common convention is as follows: Each command-line argument must be separated by a space or a tab character. Commas, semicolons, and the like are not considered separators. For example,

```
This is a test
```

is made up of four strings, but

```
this,that,and,another
```

is one string.

If you need to pass a command-line argument that does, in fact, contain spaces, you must place it between quotes, as shown in this example:

```
"this is a test"
```

The names of **argv** and **argc** are arbitrary—you can use any names you like. However, **argc** and **argv** are traditional and have been used since C's origin. It is a good idea to use these names so that anyone reading your program can quickly identify them as the command-line parameters.

One last point: the ANSI C standard only defines the **argc** and **argv** parameters. However, your compiler may allow additional parameters to **main( )**. For example, some DOS or Windows compatible compilers allow access to environmental information through a command-line argument. Check your compiler's user manual.

EXAMPLES

1. When you pass numeric data to a program, that data will be received in its string form. Your program will need to convert it into the proper internal format using one or another of C's standard library functions. The most common conversion functions are shown here, using their prototypes:

   ```
   int atoi(char *str);

   double atof(char *str);

   long atol(char *str);
   ```

   These functions use the STDLIB.H header file. The **atoi( )** function returns the **int** equivalent of its string argument. The

atof( ) returns the **double** equivalent of its string argument, and the **atol( )** returns the **long** equivalent of its string argument. If you call one of these functions with a string that is not a valid number, zero will be returned. The following program demonstrates these functions. To use it, enter an integer, a long integer, and a floating-point number on the command line. It will then redisplay them on the screen.

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  int i;
  double d;
  long l;

  i = atoi(argv[1]);
  l = atol(argv[2]);
  d = atof(argv[3]);

  printf("%d %ld %f", i, l, d);

  return 0;
}
```

2. This program coverts ounces to pounds. To use it, specify the number of ounces on the command line.

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  double pounds;

  pounds = atof(argv[1]) / 16.0;
  printf("%f pounds", pounds);

  return 0;
}
```

3. Although the examples up to this point haven't done so, you should verify in real programs, that the right number of

command-line arguments have been supplied by the user. The way to do this is to test the value of **argc**. For example, the ounces-to-pounds program can be improved as shown here:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  double pounds;

  if(argc!=2) {
    printf("Usage: CONVERT <ounces>\n");
    printf("Try Again");
  }
  else {
    pounds = atof(argv[1]) / 16.0;
    printf("%f pounds", pounds);
  }

  return 0;
}
```

This way the program will perform a conversion only if a command-line argument is present. (Of course, you may prompt the user for any missing information, if you choose.)

Generally, the preceding program will be written by a professional C programmer like this:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  double pounds;

  if(argc!=2) {
    printf("Usage: CONVERT <ounces>\n");
    printf("Try Again");
    exit(1); /* stop the program */
  }

  pounds = atof(argv[1]) / 16.0;
  printf("%f pounds", pounds);
```

```
  return 0;
}
```

When some condition necessary for a program's execution has not been met, most C programmers call the standard library function **exit( )** to terminate the program. The **exit( )** function has this prototype:

void exit(int *return-code*);

and uses the STDLIB.H header file. When **exit( )** terminates the program, it returns the value of *return-code* to the operating system. By convention, most operating systems use a return code of zero to mean that a program has terminated normally. Nonzero values indicate abnormal termination.

EXERCISES

1. Write a program that accepts two command-line arguments. Have the program compare them and report which is lexicographically greater than the other.
2. Write a program that takes two numeric arguments and displays their sum.
3. Expand the program in Exercise 2 so that it takes three arguments. The first argument must be one of these words: add, subtract, multiply, or divide. Based on the value of the first argument, perform the requested operation on the remaining two numeric arguments.

7.5 COMPARE OLD-STYLE TO MODERN FUNCTION PARAMETER DECLARATIONS

Early versions of C used a different parameter declaration method than has been shown in this book. This original declaration method is

now called the *old-style* or *classic form*. The form used in this book is the *modern form*. It was introduced when the ANSI C standard was created. While the modern form should be used for all new programs, you will still find examples of old-style parameter declarations in older programs and you need to be familiar with it.

The general form of the old-style parameter declaration is shown here:

type function-name(parameter1, parameter2,...parameterN)
type parameter1;
type parameter2;
.
.
.
type parameterN;
{
function-code
}

Notice that the declaration is divided into two parts. Within the parentheses, only the names of the parameters are specified. Outside the parentheses, the types and names are specified. For example, given the following modern declaration

```
float f(char ch, long size, double max)
{



}
```

the equivalent old-style declaration is

```
float f(ch, size, max)
char ch;
long size;
double max;
{
```

One other aspect of the old-style declaration is that you can specify more than one parameter after the type name. For example, this is perfectly valid:

```
myfunc(i, j, k)
int i, j, k;
{
.
.
.
```

The ANSI C standard specifies that either the old-style or the modern declaration form may be used. The reason for this is to maintain compatibility with older C programs. (There are literally millions of lines of C code still in existence that use the old-style form.) So, if you see programs in books or magazines that use the classic form, don't worry; your compiler will be able to compile them. However for all new programs, you should definitely use the modern form.

EXAMPLE

1. This program uses the old declaration form:

```
#include <stdio.h>

int area(int l, int w);

int main(void)
{
  printf("area is %d", area(10, 13));
  return 0;
}

int area(l, w)
int l, w;
{
  return l * w;
}
```

Notice that even though the old form of parameter declaration is used to define the function, it is still possible to prototype the function.

EXERCISES

1. Convert this program so that **f_to_m( )** uses the old-style declaration form.

```
#include <stdio.h>

double f_to_m(double f);

int main(void)
{
  double feet;

  printf("Enter feet: ");
  scanf("%lf", &feet);
  printf("Meters: %f", f_to_m(feet));

  return 0;
}

double f_to_m(double f)
{
  return f / 3.28;
}
```

At this point you should be able to answer these questions and perform these exercises:

1. How do you prototype a function that does not have parameters?
2. What is a function prototype, and what are the benefits of it?
3. How do command-line arguments get passed to a C program?

4. Write a program that uses a recursive function to display the letters of the alphabet.
5. Write a program that takes a string as a command-line argument. Have it output the string in coded form. To code the string, add 1 to each character.
6. What is the prototype for this function?

```
double myfunc(int x, int y, char ch)
{
    .
    .
    .
}
```

7. Show how the function in Exercise 6 would be coded using the old-style function declaration.
8. What does the **exit( )** function do?
9. What does the **atoi( )** function do?

This section checks how well you have integrated the material in this chapter with that from earlier chapters.

1. Write a program that allows access only if the user enters the correct password as a command-line parameter. If the user enters the right word, print **Access Permitted**; otherwise print **Access Denied**.
2. Create a function called **string_up( )** that transforms the string it is called with into uppercase characters. Demonstrate its use in a program. (Hint, use the **toupper( )** function to convert lowercase characters into uppercase.)

3. Write a function called **avg( )** that averages a list of floating-point values. The function will have two arguments. The first is a pointer to the array containing the numbers; the second is an integer value, which specifies the size of the array. Demonstrate its use in a program.
4. Explain how pointers allow C to construct a call-by-reference parameter.

8

Console I/O

chapter objectives

8.1 Learn another preprocessor directive

8.2 Examine character and string input and output

8.3 Examine some non-standard console functions

8.4 Take a closer look at **gets( )** and **puts( )**.

8.5 Master **printf( )**

8.6 Master **scanf( )**

IN this chapter you will learn about C's console I/O functions. These are the functions that read or write information to and from the console. You have already been using some of these functions. Here we will look at them in detail. This chapter begins with a short but necessary digression that introduces another of C's preprocessor directives: **#define**.

Before proceeding, you should be able to answer these questions and perform these exercises:

1. What must you do to enable the compiler to check that a function is being called correctly?
2. What are the principal advantages of using function prototypes?
3. Write a program that uses a function called **hypot( )** that returns the length of the hypotenuse of a right triangle when passed the length of the two opposing sides. Have the function return a **double** value. The type of the parameters must be **double** as well. Demonstrate the function in a program. (The Pythagorean theorem states that the sum of the squares of the two opposing sides equals the square of the hypotenuse.)
4. What return type should you use for a function that returns no value?
5. Write a recursive function called **rstrlen( )** that uses recursion to compute the length of a string. Demonstrate it in a program.
6. Write a program that reports how many command line arguments it has been called with. Also, have it display the contents of the last one.
7. How is this declaration coded using the old-style function declaration form?

```
void func(int a, char ch, double d)
```

8.1 LEARN ANOTHER PREPROCESSOR DIRECTIVE

As you recall, the C preprocessor performs various manipulations on the source code of your program before it is actually compiled. A preprocessor directive is simply an instruction to the preprocessor. Up to this point, you have learned about and have used one preprocessor directive, **#include**. Before proceeding, you need to learn about another: **#define**.

The **#define** directive tells the preprocessor to perform a text substitution throughout your entire program. That is, it causes one sequence of characters to be replaced by another. This process is generally referred to as *macro substitution*. The general form of the **#define** statement is shown here:

#define *macro-name character-sequence*

Notice that this line does not end in a semicolon. Each time the *macro-name* is encountered in the program, the associated *character-sequence* is substituted for it. For example, consider this program:

```
#include <stdio.h>

#define MAX 100

int main(void)
{
  int i;

  for(i=0; i<MAX; i++) printf("%d ", i);

  return 0;
}
```

When the identifier **MAX** is encountered by the preprocessor, **100** is automatically substituted. Thus, the **for** loop will actually look like this to the compiler:

```
for(i=0; i<100; i++) printf("%d ", i);
```

Keep one thing clearly in mind: At the time of the substitution, **100** is simply a string of characters composed of a 1 and two 0s. The

preprocessor does not convert a numeric string into its internal binary format. This is left to the compiler.

The macro name can be any valid C identifier. Thus, macro names must follow the same naming rules as do variables. Although macro names can appear in either upper- or lowercase letters, most programmers have adopted the convention of using uppercase for macro names. This makes it easy for anyone reading your program to know when a macro name is being used.

There must be one or more spaces between the macro name and the character sequence. The character sequence can contain any type of character, including spaces. It is terminated by the end of the line.

Preprocessor directives in general and **#define** in particular are not affected by C's code blocks. That is, whether you define a macro name outside of all functions or within a function, once it is defined, all code after that point may have access to it. For example, this program prints **186000** on the screen.

```
#include <stdio.h>

void f(void);

int main(void)
{
  #define LIGHTSPEED 186000

  f();

  return 0;
}

void f(void)
{
  printf("%ld", LIGHTSPEED);
}
```

There is one important point you must remember: Each preprocessor directive must appear on its own line.

Macro substitutions are useful for two main reasons. First, many C library functions use certain predefined values to indicate special conditions or results. Your programs will need access to these values when they use one of these functions. However, many times the actual value will vary between programming environments. For this

reason, these values are usually specified using macro names. The macro names are defined inside the header file that relates to each specific function. You will see an example of this in the next section.

The second reason macro substitution is important is that it can help make it easier to maintain programs. For example, if you know that a value, such as an array size, is going to be used several places in your program, it is better to create a macro for this value. Then if you ever need to change this value, you simply change the macro definition. All references to it will be changed automatically when the program is recompiled.

EXAMPLES

1. Since a macro substitution is simply a text replacement, you can use a macro name in place of a quoted string. For example, the following program prints **Macro Substitutions are Fun**.

```
#include <stdio.h>

#define FUN "Macro Substitutions are Fun"

int main(void)
{
  printf(FUN);

  return 0;
}
```

To the compiler, the **printf( )** statement looks like this:

```
printf("Macro Substitutions are Fun");
```

2. Once a macro name has been defined, it can be used to help define another macro name. For example, consider this program:

```
#include <stdio.h>

#define SMALL 1
#define MEDIUM SMALL+1
#define LARGE MEDIUM+1

int main(void)
{
  printf("%d %d %d", SMALL, MEDIUM, LARGE);
```

```
  return 0;
}
```

As you might expect, it prints **1 2 3** on the screen.

3. If a macro name appears inside a quoted string, no substitution will take place. For example, given this definition

```
#define ERROR "catastrophic error occurred"
```

the following statement will not be affected.

```
printf("ERROR: Try again");
```

EXERCISES

1. Create a program that defines two macro names, **MAX** and **COUNTBY**. Have the program count from zero to **MAX-1** by whatever value **COUNTBY** is defined as. (Give **COUNTBY** the value 3 for demonstration purposes.)
2. Is this fragment correct?

```
#define MAX MIN+100
#define MIN 10
```

3. Is this fragment correct?

```
#define STR this is a test

.
.
printf(STR);
```

4. Is this program correct?

```
#define STDIO <stdio.h>
#include STDIO

int main(void)
{
  printf("This is a test.");

  return 0;
}
```

.2 EXAMINE CHARACTER AND STRING INPUT AND OUTPUT

Although you have already learned how to input and output characters and strings, this section looks at these processes more formally.

The ANSI C standard defines these two functions that perform character input and output, respectively:

```
int getchar(void);
int putchar(int ch);
```

They both use the header file STDIO.H. As mentioned earlier in this book, many compilers implement **getchar( )** in a line-buffered manner, which makes its use limited in an interactive environment. Most compilers contain a non-standard function called **getche( )**, which operates like **getchar( )**, except that it is interactive. Discussion of **getche( )** and other non-standard functions will occur in a later section.

The **getchar( )** function returns the next character typed on the keyboard. This character is read as an **unsigned char** converted to an **int**. However, most commonly, your program will assign this value to a **char** variable, even though **getchar( )** is declared as returning an **int**. If you do this, the high-order byte(s) of the integer is simply discarded.

The reason that **getchar( )** returns an integer is that when an error occurs while reading input, **getchar( )** returns the macro **EOF**, which is a negative integer (usually -1). The **EOF** macro, defined in STDIO.H, stands for end-of-file. Since **EOF** is an integer value, to allow it to be returned, **getchar( )** must return an integer. In the vast majority of circumstances, if an error occurs when reading from the keyboard, it means that the computer has ceased to function. Therefore, most programmers don't usually bother checking for **EOF** when using **getchar( )**. They just assume a valid character has been returned. Of course, there are circumstances in which this is not appropriate—for example, when I/O is redirected, as explained in Chapter 9. But most of the time you will not need to worry about **getchar( )** encountering an error.

The **putchar( )** function outputs a single character to the screen. Although its parameter is declared to be of type **int**, this value is converted into an **unsigned char** by the function. Thus, only the

low-order byte of *ch* is actually displayed. If the output operation is successful, **putchar( )** returns the character written. If an output error occurs, **EOF** is returned. For reasons similar to those given for **getchar( )**, if output to the screen fails, the computer has probably crashed anyway, so most programmers don't bother checking the return value of **putchar( )** for errors.

The reason you might want to use **putchar( )** rather than **printf( )** with the **%c** specifier to output a character is that **putchar( )** is faster and more efficient. Because **printf( )** is more powerful and flexible, a call to **printf( )** generates greater overhead than a call to **putchar( )**.

EXAMPLES

1. As stated earlier, **getchar( )** is generally implemented using line buffering. When input is line buffered, no characters are actually passed back to the calling program until the user presses ENTER. The following program demonstrates this:

```
#include <stdio.h>

int main(void)
{
  char ch;
  do {
    ch = getchar();
    putchar('.');
  } while(ch != '\n');

  return 0;
}
```

   Instead of printing a period between each character, what you will see on the screen is all the letters you typed before pressing ENTER, followed by a string of periods.

   One other point: When entering characters using **getchar( )**, pressing ENTER will cause the newline character (**\n**) to be returned. However, when using one of the alternative non-standard functions, pressing ENTER will cause the carriage return character (**\r**) to be returned. Keep this difference in mind.

2. The following program illustrates the fact that you can use C's backslash character constants with **putchar( )**.

```
#include <stdio.h>

int main(void)
{
  putchar('A');
  putchar('\n');
  putchar('B');

  return 0;
}
```

This program displays

```
A
B
```

on the screen.

EXERCISES

1. Rewrite the program shown in the first example so that it checks for errors on both input and output operations.
2. What is wrong with this fragment?

```
char str[80] = "this is a test";
.
.
.
putchar(str);
```

8.3 EXAMINE SOME NON-STANDARD CONSOLE FUNCTIONS

Because character input using **getchar( )** is usually line-buffered, many compilers supply additional input routines that provide interactive character input. You have already been introduced to one of these: **getche( )**. Here is its prototype and that of its close relative **getch( )**:

```
int getche(void);
int getch(void);
```

Both functions use the header file CONIO.H. The **getche( )** function waits until the next keystroke is entered at the keyboard. When a key is pressed, **getche( )** echoes it to the screen and then immediately returns the character. The character is read as an **unsigned char** and elevated to **int**. However, your routines can simply assign this value to a **char** value. The **getch( )** function is the same as **getche( )**, except that the keystroke is not echoed to the screen.

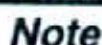
Another very useful non-ANSI-standard function commonly supplied with a C compiler is **kbhit( )**. It has this prototype:

```
int kbhit(void);
```

The **kbhit( )** function also requires the header file CONIO.H. This function is used to determine whether a key has been pressed or not. If the user has pressed a key, this function returns true (nonzero), but does not read the character. If a keystroke is waiting, you may read it with **getche( )** or **getch( )**. If no keystroke is pending, **kbhit( )** returns false (zero).

For some compilers, the non-standard I/O functions such as **getche( )** are not compatible with the standard I/O functions such as **printf( )** or **scanf( )**. When this is the case, mixing the two can cause unusual program behavior. Most troubles caused by this incompatibility occur when inputting information (although problems could occur on output). If the standard and non-standard I/O functions are not compatible in your compiler, you may need to use non-standard versions of **scanf( )** and/or **printf( )**, too. These are called **cprintf( )** and **cscanf( )**.

The **cprintf( )** function works like **printf( )** except that it does not translate the newline character (**\n**) into the carriage return, linefeed pair as does the **printf( )** function. Therefore, it is necessary to explicitly output the carriage return (**\r**) where desired. The **cscanf( )** function works like the **scanf( )** function. Both **cprintf( )** and **cscanf( )** use the CONIO.H header file. The **cprintf( )** and **cscanf( )** functions are expressly designed to be compatible with **getch( )** and **getche( )**, as well as other non-standard I/O functions.



Microsoft C++ supports the functions just described. In addition, it provides alternative names for the functions that begin with an underscore. For example, when using Visual C++, you can specify ***getche( )*** *as* ***_getche( )****, too.*

One last point: Even for compilers that have incompatibilities between the standard and non-standard I/O functions, such incompatibilities sometimes only apply in one case and not another. If you encounter a problem, just try substituting a different function.

EXAMPLES

1. The **getch( )** function lets you take greater control of the screen because you can determine what is displayed each time a key is struck. For example, this program reads characters until a 'q' is typed. All characters are displayed in uppercase using the **cprintf( )** function.

```
#include <stdio.h>
#include <conio.h>
#include <ctype.h>

int main(void)
{
  char ch;

  do {
    ch = getch();
    cprintf("%c", toupper(ch));
  } while(ch != 'q');

  return 0;
}
```

2. The **kbhit( )** function is very useful when you want to let a user interrupt a routine without actually forcing the user to continually respond to a prompt like "Continue?". For example, this program prints a 5-percent sales-tax table in increments of 20 cents. The program continues to print the table until either the user strikes a key or the maximum value is printed.

```
#include <stdio.h>
#include <conio.h>

int main(void)
{
  double amount;
```

```
    amount = 0.20;

    cprintf("Printing 5-percent tax table\n\r");
    cprintf("Press a key to stop.\n\n\r");
    do {
      cprintf("amount: %f, tax: %f\n\r", amount,
              amount*0.05);
      if(kbhit()) break;
      amount = amount + 0.20;
    } while(amount < 100.0);

    return 0;
}
```

In the calls to **cprintf( )**, notice how both the carriage return (**r**) and the newline (**n**) must be output. As explained, **cprintf( )** does not automatically convert newlines into carriage return, linefeed pairs.

EXERCISES

1. Write a program that displays the ASCII code of each character typed. Do not display the actual character, however.
2. Write a program that prints periods on the screen until you press a key.

8.4 TAKE A CLOSER LOOK AT gets( ) AND puts( )

Although both **gets( )** and **puts( )** were introduced earlier, let's take a closer look at them now. Their function prototypes are

```
char *gets(char *str);
int puts(char *str);
```

These functions use the header file STDIO.H. The **gets( )** function reads characters entered at the keyboard until a carriage return is read (i.e., until the user presses ENTER). It stores the characters in the array

pointed to by *str*. The carriage return is not added to the string. Instead, it is converted into the null terminator. If successful, **gets( )** returns a pointer to the start of *str*. If an error occurs, a null pointer is returned.

The **puts( )** function outputs the string pointed to by *str* to the screen. It automatically appends a carriage return, line-feed sequence. If successful, **puts( )** returns a non-negative value. If an error occurs, **EOF** is returned.

The main reason you may want to use **puts( )** instead of **printf( )** to output a string is that **puts( )** is much smaller and faster. While this is not important in the example programs shown in this book, it may be in some applications.

EXAMPLES

1. This program shows how you can use the return value of **gets( )** to access the string holding the input information. Notice that this program also confirms that no error has occurred before attempting to use the string.

```
#include <stdio.h>

int main(void)
{
  char *p, str[80];

  printf("Enter a string: ");
  p = gets(str);
  if(p) /* if not null */
    printf("%s %s", p, str);

  return 0;
}
```

2. If you simply want to make sure that **gets( )** did not encounter an error before proceeding, you can place **gets( )** directly inside an **if** statement, as illustrated by the following program:

```
#include <stdio.h>

int main(void)
{
  char str[80];
```

```
  printf("Enter a string: ");
  if(gets(str))  /* if not null */
    printf("Here is your string: %s", str);

  return 0;
}
```

Because a null pointer is false, there is no need for the intermediary variable **p**, and the **gets( )** statement can be put directly inside the **if**.

3. It is important to understand that even though **gets( )** returns a pointer to the start of the string, it still must be called with a pointer to an actual array. For example, the following is wrong:

```
char *p;

p = gets(p); /* wrong!!! */
```

Here, there is no array defined into which **gets( )** can put the string. This will result in a program failure.

4. This program outputs the words **one**, **two**, and **three** on three separate lines, using **puts( )**.

```
#include <stdio.h>

int main(void)
{
  puts("one");
  puts("two");
  puts("three");

  return 0;
}
```

EXERCISES

1. Compile the program shown in Example 2, above. Note the size of the compiled code. Next, convert it so that it uses **printf( )** statements, instead of **puts( )**. You will find that the **printf( )** version is several bytes larger.

2. Is this program correct? If not, why not?

```
#include <stdio.h>

int main(void)
{
  char *p, *q;

  printf("Enter a string: ");
  p = gets(q);
  printf(p);

  return 0;
}
```

8.5 MASTER printf( )

Although you already know many things about **printf( )**, you will be surprised by how many more features it has. In this section you will learn about some more of them. To begin, let's review what you know so far.

The **printf( )** function has this prototype:

int printf(char *control-string, ...);

The periods indicate a variable-length argument list. The **printf( )** function returns the number of characters output. If an error occurs, it returns a negative number. Frankly, few programmers bother with the return value of **printf( )** because, as mentioned earlier, if the console is not working, the computer is probably not functional anyway.

The control string may contain two types of items: characters to be output and format specifiers. All format specifiers begin with **%**. A *format specifier*, also referred to as a *format code*, determines how its matching argument will be displayed. Format specifiers and their arguments are matched from left to right, and there must be as many arguments as there are specifiers.

The format specifiers accepted by **printf( )** are shown in Table 8-1. You have already learned about the **%c**, **%d**, **%s**, **%u**, **%p**, and **%f** specifiers. The others will be examined now.

Code	Format
%c	Character
%d	Signed decimal integers
%i	Signed decimal integers
%e	Scientific notation (lowercase 'e')
%E	Scientific notation (uppercase 'E')
%f	Decimal floating point
%g	Uses %e or %f, whichever is shorter
%G	Uses %E or %f, whichever is shorter
%o	Unsigned octal
%s	String of characters
%u	Unsigned decimal integers
%x	Unsigned hexadecimal (lowercase letters)
%X	Unsigned hexadecimal (uppercase letters)
%p	Displays a pointer
%n	The associated argument is a pointer to an integer into which the number of characters written so far is placed.
%%	Prints a % sign

TABLE 8-1 *The* **printf( )** *Format Specifiers*

The **%i** command is the same as **%d** and is redundant.

You can display numbers of type **float** or **double** using scientific notation by using either **%e** or **%E**. The only difference between the two is that **%e** uses a lowercase 'e' and **%E** uses an uppercase 'E'. These specifiers may have the **L** modifier applied to them to allow them to display values of type **long double**.

The **%g** and **%G** specifiers cause output to be in either normal or scientific notation, depending upon which is shorter. The difference between the **%g** and the **%G** is whether a lower- or uppercase 'e' is used in cases where scientific notation is shorter. These specifiers may have the **L** modifier applied to them to allow them to display values of type **long double**.

You can display an integer in octal format using **%o** or in hexadecimal using **%x** or **%X**. Using **%x** causes the letters 'a' through 'f' to be displayed in lowercase. Using **%X** causes them to be displayed in uppercase. These specifiers may have the **h** and **l** modifiers applied to allow them to display **short** and **long** data types, respectively.

The argument that matches the **%n** specifier must be a pointer to an integer. When the **%n** is encountered, **printf( )** assigns the integer pointed to by the associated argument the number of characters output so far.

Since all format commands begin with a percent sign, you must use **%%** to output a percent sign.

All but the **%%**, **%p**, and **%c** specifiers may have a minimum-field-width specifier and/or a precision specifier associated with them. Both of these are integer quantities. If the item to output is shorter than the specified minimum field width, the output is padded with spaces, so that it equals the minimum width. However, if the output is longer than the minimum, output is *not* truncated. The minimum-field-width specifier is placed after the **%** sign and before the format specifier.

The precision specifier follows the minimum-field-width specifier. The two are separated by a period. The precision specifier affects different types of format specifiers differently. If it is applied to the **%d**, **%i**, **%o**, **%u** or **%x** specifiers, it determines how many digits are to be shown. Leading zeros are added if needed. When applied to **%f**, **%e**, or **%E**, it determines how many digits will be displayed after the decimal point. For **%g** or **%G**, it determines the number of significant digits. When applied to the **%s**, it specifies a maximum field width. If a string is longer than the maximum-field-width specifier, it will be truncated.

By default, all numeric output is right justified. To left justify output, put a minus sign directly after the **%** sign.

The general form of a format specifier is shown here. Optional items are shown between brackets.

%[-][*minimum-field-width*][.][*precision*]*format-specifier*

For example, this format specifier tells **printf( )** to output a **double** value using a field width of 15, with 2 digits after the decimal point.

```
%15.2f
```

EXAMPLES

1. If you don't want to specify a minimum field width, you can still specify the precision. Simply put a period in front of the precision value, as illustrated by the following program:

```
#include <stdio.h>

int main(void)
{
  printf("%.5d\n", 10);
  printf("$%.2f\n", 99.95);
  printf("%.10s", "Not all of this will be printed\n");

  return 0;
}
```

The output from this program looks like this:

```
00010
$99.95
Not all of
```

Notice the effect of the precision specifier as applied to each data type.

2. The minimum-field-width specifier is especially useful for creating tables that contain columns of numbers that must line up. For example, this program prints 1000 random numbers in three columns. It uses another of C's standard library functions, **rand( )**, to generate the random numbers. The **rand( )** function returns a random integer value each time it is called. It uses the header STDLIB.H.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  int i;

  for(i=0; i<1000; i++)
    printf("%10d %10d %10d\n", rand(), rand(), rand());

  return 0;
}
```

Part of the output from this program is shown here. Notice how the columns are aligned. (Remember, if you try the program, you will probably see different numbers.)

```
10982     130     346
 7117   11656    1090
22948    6415   17595
14558    9004   31126
18492   22879    3571
26721    5412    1360
27119   25047   22463
13985    7190   31441
30252   27509   31214
19816   14779   26571
17995   19651   21681
13310    3734   23593
15561   21995    3979
11288   18489   16092
 5892    8664   28466
 5364   22766   13863
20427   21151   17639
 8812   25795     100
12347   12666   15108
```

3. This program prints the value 90 four different ways: decimal, octal, lowercase hexadecimal, and uppercase hexadecimal. It also prints a floating-point number using scientific notation with a lowercase 'e' and an uppercase 'E'.

```
#include <stdio.h>

int main(void)
{
  printf("%d %o %x %X\n", 90, 90, 90, 90);
  printf("%e %E\n", 99.231, 99.231);

  return 0;
}
```

The output from this program is shown here:

```
90 132 5a 5A
9.92310e+01 9.92310E+01
```

4. The following program demonstrates the **%n** specifier:

```
#include <stdio.h>

int main(void)
{
```

Like **printf( )**, **scanf( )** has many more features than we have used so far. In this section, several of these additional features are explored. Let's begin by reviewing what you have already learned.

The prototype for **scanf( )** is shown here:

int scanf(char **control-string*, ...);

The *control-string* consists mostly of format specifiers. However, it can contain other characters. (You will learn about the effect of other characters in the control string soon.) The format specifiers determine

8.6 MASTER scanf( )

```
  int i;

  printf("%d %f\n%n", 100, 123.23, &i);
  printf("%d characters output so far", i);

  return 0;
}
```

Its output looks like this:

```
100 123.230000
15 characters output so far
```

The fifteenth character is the newline.

EXERCISES

1. Write a program that prints a table of numbers, each line consisting of a number, its square, and its cube. Have the table begin at 2 and end at 100. Make the columns line up, and left justify each column.
2. How would you output this line using **printf( )**?

   Clearance price: 40% off as marked

3. Show how to display **1023.03** so that only two decimal places are printed.

how **scanf( )** reads information into the variables pointed to by the arguments that follow the control string. The specifiers are matched in order, from left to right, with the arguments. There must be as many arguments as there are specifiers. The format specifiers are shown in Table 8-2. As you can see, the **scanf( )** specifiers are very much like the **printf( )** specifiers.

The **scanf( )** function returns the number of fields assigned values. If an error occurs before any assignments are made, **EOF** is returned.

The specifiers **%x** and **%o** are used to read an unsigned integer using hexadecimal and octal number bases, respectively.

The specifiers **%d**, **%i**, **%u**, **%x**, and **%o** may be modified by the **h** when inputting into a **short** variable and by **l** when inputting into a **long** variable.

The specifiers **%e**, **%f**, and **%g** are equivalent. They all read floating-point numbers represented in either scientific notation or standard decimal notation. Unmodified, they input information into a **float** variable. You can modify them using an **l** when inputting into a **double**. To read a **long double**, modify them with an **L**.

You can use **scanf( )** to read a string using the **%s** specifier, but you probably won't want to. Here's why: When **scanf( )** inputs a string, it stops reading that string when the first whitespace character is encountered. A whitespace character is either a space, a tab, or a

Code	Meaning
%c	Read a single character
%d	Read a decimal integer
%i	Read a decimal integer
%e	Read a floating-point number
%f	Read a floating-point number
%g	Read a floating-point number
%o	Read an octal number
%s	Read a string
%x	Read a hexadecimal number
%p	Read a pointer
%n	Receives an integer value equal to the number of characters read so far
%u	Read an unsigned integer
%[ ]	Scan for a set of characters

TABLE 8-2 *The* **scanf( )** *Format Specifiers*

newline. This means that you cannot easily use **scanf( )** to read input like this into a string:

```
this is one string
```

Because there is a space after "this," **scanf( )** will stop inputting the string at that point. This is why **gets( )** is generally used to input strings.

The **%p** specifier inputs a memory address using the format determined by the host environment. The **%n** specifier assigns the number of characters input up to the point the **%n** is encountered to the integer variable pointed to by its matching argument. The **%n** may be modified by either **l** or **h** so that it may assign its value to either a **long** or **short** variable.

A very interesting feature of **scanf( )** is called a *scanset*. A scanset specifier is created by putting a list of characters inside square brackets. For example, here is a scanset specifier containing the letters 'ABC.'

```
%[ABC]
```

When **scanf( )** encounters a scanset, it begins reading input into the character array pointed to by the scanset's matching argument. It will only continue reading characters as long as the next character is part of the scanset. As soon as a character that is not part of the scanset is found, **scanf( )** stops reading input for this specifier and moves on to any others in the control string.

You can specify a range in a scanset using the - (hyphen). For example, this scanset specifies the characters 'A' through 'Z'.

```
%[A-Z]
```

Technically, the use of the hyphen to specify a range is not specified by the ANSI C standard, but it is nearly universally accepted.

When the scanset is very large, sometimes it is easier to specify what is *not* part of a scanset. To do this, precede the set with a ^. For example,

```
%[^0123456789]
```

When **scanf( )** encounters this scanset, it will read any characters *except* the digits 0 through 9.

You can suppress the assignment of a field by putting an asterisk immediately after the % sign. This can be very useful when inputting information that contains needless characters. For example, given this **scanf( )** statement

```
int first, second;
scanf("%d%*c%d", &first, &second);
```

this input

```
555-2345
```

will cause **scanf( )** to assign **555** to **first**, discard the -, and assign **2345** to **second**. Since the hyphen is not needed, there is no reason to assign it to anything. Hence, no associated argument is supplied.

You can specify a maximum field width for all specifiers except **%c**, for which a field is always one character, and **%n**, to which the concept does not apply. The maximum field width is specified as an unsigned integer, and it immediately precedes the format specifier character. For example, this limits the maximum length of a string assigned to **str** to 20 characters:

```
scanf("%20s", str);
```

If a space appears in the control string, then **scanf( )** will begin reading and discarding whitespace characters until the first non-whitespace character is encountered. If any other character appears in the control string, **scanf( )** reads and discards all matching characters until it reads the first character that does not match that character.

One other point: As **scanf( )** is generally implemented, it line-buffers input in the same way that **getchar( )** often does. While this makes little difference when inputting numbers, its lack of interactivity tends to make **scanf( )** of limited value for other types of input.

EXAMPLES

1. To see the effect of the **%s** specifier, try this program. When prompted, type **this is a test** and press ENTER. You will see only **this** redisplayed on the screen. This is because, when reading strings, **scanf( )** stops when it encounters the first whitespace character.

```
#include <stdio.h>

int main(void)
{
  char str[80];

  /* Enter "this is a test" */
  printf("Enter a string: ");
  scanf("%s", str);
  printf(str);

  return 0;
}
```

2. Here's an example of a scanset that accepts both the upper- and lowercase characters. Try entering some letters, then any other character, and then some more letters. After you press ENTER, only the letters that you entered before pressing the non-letter key will be contained in **str**.

```
#include <stdio.h>

int main(void)
{
  char str[80];

  printf("Enter letters, anything else to stop\n");
  scanf("%[a-zA-Z]", str);

  printf(str);

  return 0;
}
```

3. If you want to read a string containing spaces using **scanf( )**, you can do so using the scanset shown in this slight variation of the previous program.

```
#include <stdio.h>

int main(void)
{
  char str[80];

  printf("Enter letters and spaces\n");
```

```
  scanf("%[a-zA-Z ]", str);
  printf(str);

  return 0;
}
```

You could also specify punctuation symbols and digits, so that you can read virtually any type of string. However, this is a fairly cumbersome way of doing things.

4. This program lets the user enter a number followed by an operator followed by a second number, such as 12+4. It then performs the specified operation on the two numbers and displays the results.

```
#include <stdio.h>

int main(void)
{
  int i, j;
  char op;

  printf("Enter operation: ");
  scanf("%d%c%d", &i, &op, &j);

  switch(op) {
    case '+': printf("%d", i+j);
      break;
    case '-': printf("%d", i-j);
      break;
    case '/': if(j) printf("%d", i/j);
      break;
    case '*': printf("%d", i*j);
  }

  return 0;
}
```

Notice that the format for entering the information is somewhat restricted because no spaces are allowed between the first number and the operator. It is possible to remove this restriction. As you know, **scanf( )** automatically discards leading whitespace characters except when you use the **%c** specifier. However, since you know that the operator will not be

a whitespace character, you can modify the **scanf( )** command to look like this:

```
scanf("%d %c%d", &i, &op, &j);
```

Whenever there is a space in the control string, **scanf( )** will match and discard whitespace characters until the first non-whitespace character is found. This includes matching zero whitespace characters. With this change in place, you can enter the information into the program using one or more spaces between the first number and the operator.

5. This program illustrates the maximum-field-width specifier:

```
#include <stdio.h>

int main(void)
{
  int i, j;

  printf("Enter an integer: ");
  scanf("%3d%d", &i, &j);
  printf("%d %d", i, j);

  return 0;
}
```

If you run this program and enter the number **12345**, **i** will be assigned 123, and **j** will have the value 45. The reason for this is that **scanf( )** is told that **i**'s field is only three characters long. The remainder of the input is then sent to **j**.

6. This program illustrates the effect of having non-whitespace characters in the control string. It allows you to enter a decimal value, but it assigns the digits to the left of the decimal point to one integer and those to the right of the decimal to another. The decimal point between the two **%d** specifiers causes the decimal point in the number to be matched and discarded.

```
#include <stdio.h>

int main(void)
{
  int i, j;

  printf("Enter a decimal number: ");
```

```
  scanf("%d.%d", &i, &j);
  printf("left part: %d, right part: %d", i, j);

  return 0;
}
```

EXERCISES

1. Write a program that prompts for your name and then inputs your first, middle, and last names. Have the program read no more than 20 characters for each part of your name. Finally, have the program redisplay your name.
2. Write a program that reads a floating-point number as a string using a scanset.
3. Is this fragment correct? If not why not?

   ```
   char ch;

   scanf("%2c", &ch);
   ```

4. Write a program that inputs a string, a **double**, and an integer. After these items have been read, have the program display how many characters were input. (Hint: use the **%n** specifier.)
5. Write a program that converts a hexadecimal number entered by the user into its corresponding decimal and octal equivalents.

Mastery Skills Check

Before proceeding you should be able to answer these questions and perform these exercises:

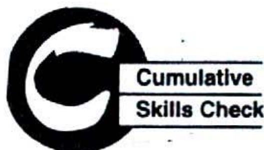
1. What is the difference between **getchar( )**, **getche( )**, and **getch( )**?
2. What is the difference between the **%e** and the **%E printf( )** format specifiers?

This section checks how well you have integrated the material in this chapter with that from earlier chapters.

1. Write a program that allows you to enter the batting averages for the players on a little league team. (Assume there are exactly 9 players.) Have the user enter the first name and batting average of each player. Use a two-dimensional character array to hold the names and a one-dimensional **double** array to hold the batting averages. Once all the names are entered, have the program report the name and average of the players with the highest and lowest averages. Also, have the program display the team average.

2. Write a program that is a simple electronic library card catalog. Have the program display this menu:

   Card Catalog:
   1. Enter
   2. Search by Author
   3. Search by Title
   4. Quit
   Choose your selection:



3. What is a scanset?

4. Write a program, using **scanf( )**, that inputs your first name, birth date (using the format mm/dd/yy), and telephone number. Redisplay the information on the screen to verify that it was input correctly.

5. What is one advantage to using **puts( )** over **printf( )** when you only need to output a string? What is one disadvantage to **puts( )**?

6. Write a program that defines a macro called **COUNT** as the value 100. Have the program then use this macro to control a **for** loop that displays the numbers 0 through 99.

7. What is **EOF**, and where is it defined?

If you choose Enter, have the program repeatedly input the name, author, and publisher of a book. Have this process continue until the user enters a blank line for the name of the book.

For searches, prompt the user for the specified author or title and then, if a match is found, display the rest of the information. After you finish this program, keep your file, because in the next chapter you will learn how to save the catalog to a disk file.