Chapter 7

Functions

We have already seen that C supports the use of library functions, which are used to carry out a number of commonly used operations or calculations (see Sec. 3.6). However, C also allows programmers to define their own functions for carrying out various individual tasks. This chapter concentrates on the creation and utilization of such programmer-defined functions.

The use of programmer-defined functions allows a large program to be broken down into a number of smaller, self-contained components, each of which has some unique, identifiable purpose. Thus a C program can be *modularized* through the intelligent use of such functions. (C does not support other forms of modular program development, such as the procedures in Pascal or the subroutines in Fortran.)

There are several advantages to this modular approach to program development. For example, many programs require that a particular group of instructions be accessed repeatedly, from several different places within the program. The repeated instructions can be placed within a single function, which can then be accessed whenever it is needed. Moreover, a different set of data can be transferred to the function each time it is accessed. Thus, *the use of a function avoids the need for redundant (repeated) programming of the same instructions.*

Equally important is the *logical clarity* resulting from the decomposition of a program into several concise functions, where each function represents some well-defined part of the overall problem. Such programs are easier to write and easier to debug, and their logical structure is more apparent than programs which lack this type of structure. This is especially true of lengthy, complicated programs. Most C programs are therefore modularized in this manner, even though they may not involve repeated execution of the same tasks. In fact the decomposition of a program into individual program modules is generally considered to be an important part of good programming practice.

The use of functions also enables a programmer to build a *customized library* of frequently used routines or of routines containing system-dependent features. Each routine can be programmed as a separate function and stored within a special library file. If a program requires a particular routine, the corresponding library function can be accessed and attached to the program during the compilation process. Hence a single function can be utilized by many different programs. This avoids repetitive programming between programs. It also promotes *portability* since programs can be written that are independent of system-dependent features.

In this chapter we will see how functions are defined and how they are accessed from various places within a C program. We will then consider the manner in which information is passed to a function. Our discussion will include the use of *function prototypes*, as recommended by the current ANSI standard. And finally, we will discuss an interesting and important programming technique known as *recursion*, in which a function can access itself repeatedly.

7.1 A BRIEF OVERVIEW

A *function* is a self-contained program segment that carries out some specific, well-defined task. Every C program consists of one or more functions (see Sec. 1.5). One of these functions must be called `main`. Execution of the program will always begin by carrying out the instructions in `main`. Additional functions will be subordinate to `main`, and perhaps to one another.

If a program contains multiple functions, their definitions may appear in any order, though they must be independent of one another. That is, one function definition cannot be embedded within another.

A function will carry out its intended action whenever it is *accessed* (i.e., whenever the function is "called") from some other portion of the program. The same function can be accessed from several different

places within a program. Once the function has carried out its intended action, control will be returned to the point from which the function was accessed.

Generally, *a function will process information that is passed to it from the calling portion of the program, and return a single value.* Information is passed to the function via special identifiers called *arguments* (also called *parameters*), and returned via the `return` statement. Some functions, however, accept information but do not return anything (as, for example, the library function `printf`), whereas other functions (e.g., the library function `scanf`) return multiple values.

EXAMPLE 7.1 Lowercase to Uppercase Character Conversion In Example 3.31 we saw a simple C program that read in a single lowercase character, converted it to uppercase using the library function `toupper`, and then displayed the uppercase equivalent. We now consider a similar program, though we will define and utilize our own function for carrying out the lowercase to uppercase conversion.

Our purpose in doing this is to illustrate the principal features involved in the use of functions. Hence, you should concentrate on the overall logic, and not worry about the details of each individual statement just yet.

Here is the complete program.

```
/* convert a lowercase character to uppercase using a programmer-defined function */

#include <stdio.h>

char lower_to_upper(char c1)          /* function definition */

{
    char c2;

    c2 = (c1 >= 'a' && c1 <= 'z') ? ('A' + c1 - 'a') : c1;
    return(c2);
}

main()

{
    char lower, upper;

    printf("Please enter a lowercase character: ");
    scanf("%c", &lower);
    upper = lower_to_upper(lower);
    printf("\nThe uppercase equivalent is %c\n\n", upper);
}
```

This program consists of two functions—the required `main` function, preceded by the programmer-defined function `lower_to_upper`. Note that `lower_to_upper` carries out the actual character conversion. This function converts only lowercase letters; all other characters are returned intact. A lowercase letter is transferred into the function via the argument `c1`, and the uppercase equivalent, `c2`, is returned to the calling portion of the program (i.e., to `main`) via the `return` statement.

Now consider the `main` function, which follows `lower_to_upper`. This function reads in a character (which may or may not be a lowercase letter) and assigns it to the char-type variable `lower`. Function `main` then calls the function `lower_to_upper`, transferring the lowercase character (`lower`) to `lower_to_upper`, and receiving the equivalent uppercase character (`upper`) from `lower_to_upper`. The uppercase character is then displayed, and the program ends. Notice that the variables `lower` and `upper` in `main` correspond to the variables `c1` and `c2` within `lower_to_upper`.

We will consider the rules associated with function definitions and function accesses in the remainder of this chapter.

7.2 DEFINING A FUNCTION

A function definition has two principal components: the *first line* (including the *argument declarations*), and the *body* of the function.

The first line of a function definition contains the type specification of the value returned by the function, followed by the function name, and (optionally) a set of arguments, separated by commas and enclosed in parentheses. Each argument is preceded by its associated type declaration. An empty pair of parentheses must follow the function name if the function definition does not include any arguments.

In general terms, the first line can be written as

data-type name(*type 1 arg 1*, *type 2 arg 2*, . . ., *type n arg n*)

where *data-type* represents the data type of the item that is returned by the function, *name* represents the function name, and *type 1*, *type 2*, . . . , *type n* represent the data types of the arguments *arg 1*, *arg 2*, . . . , *arg n*. The data types are assumed to be of type `int` if they are not shown explicitly. However, the omission of the data types is considered poor programming practice, even if the data items are integers.

The arguments are called *formal arguments*, because they represent the names of data items that are transferred into the function from the calling portion of the program. They are also known as *parameters* or *formal parameters*. (The corresponding arguments in the function *reference* are called *actual arguments*, since they define the data items that are actually transferred. Some textbooks refer to actual arguments simply as *arguments*, or as *actual parameters*.) The identifiers used as formal arguments are "local" in the sense that they are not recognized outside of the function. Hence, the names of the formal arguments need not be the same as the names of the actual arguments in the calling portion of the program. Each formal argument must be of the same *data type*, however, as the data item it receives from the calling portion of the program.

The remainder of the function definition is a compound statement that defines the action to be taken by the function. This compound statement is sometimes referred to as the *body* of the function. Like any other compound statement, this statement can contain expression statements, other compound statements, control statements, and so on. It should include one or more `return` statements, in order to return a value to the calling portion of the program.

A function can access other functions. In fact, it can even access itself (this process is known as *recursion* and is discussed in Sec. 7.6).

EXAMPLE 7.2 Consider the function `lower_to_upper`, which was originally presented in Example 7.1.

```
char lower_to_upper(char c1)        /* programmer-defined conversion function */
{
    char c2;

    c2 = (c1 >= 'a' && c1 <= 'z') ? ('A' + c1 - 'a') : c1;
    return(c2);
}
```

The first line contains the function name, `lower_to_upper`, followed by the formal argument `c1`, enclosed in parentheses. The *function name* is preceded by the data type `char`, which describes the data item that is returned by the function. In addition, the *formal argument* `c1` is preceded by the data type `char`. This later data type, which is included within the pair of parentheses, refers to the formal argument. The formal argument, `c1`, represents the lowercase character that is transferred *to* the function from the calling portion of the program.

The body of the function begins on the second line, with the declaration of the local char-type variable `c2`. (Note the distinction between the *formal argument* `c1`, and the *local variable* `c2`.) Following the declaration of `c2` is a statement that tests whether `c1` represents a lowercase letter and then carries out the conversion. The original character is returned intact if it is not a lowercase letter. Finally, the `return` statement (see below) causes the converted character to be returned to the calling portion of the program.

Information is returned from the function to the calling portion of the program via the `return` statement. The `return` statement also causes the program logic to return to the point from which the function was accessed.

In general terms, the `return` statement is written as

```
return expression;
```

The value of the *expression* is returned to the calling portion of the program, as in Example 7.2 above. The *expression* is optional. If the *expression* is omitted, the `return` statement simply causes control to revert back to the calling portion of the program, without any transfer of information.

Only one expression can be included in the `return` statement. Thus, a function can return only one value to the calling portion of the program via `return`.

A function definition can include multiple `return` statements, each containing a different expression. Functions that include multiple branches often require multiple returns.

EXAMPLE 7.3 Here is a variation of the function `lower_to_upper`, which appeared in Examples 7.1 and 7.2.

```
char lower_to_upper(char c1)        /* programmer-defined conversion function */
{
    if (c1 >= 'a' && c1 <= 'z')
       return('A' + c1 - 'a');
    else
       return(c1);
}
```

This function utilizes the `if - else` statement rather than the conditional operator. It is somewhat less compact than the original version, though the logic is clearer. In addition, note that this form of the function does not require the local variable `c2`.

This particular function contains two different `return` statements. The first returns an expression that represents the uppercase equivalent of the lowercase character ; the second returns the original lowercase character, unchanged.

The `return` statement can be absent altogether from a function definition, though this is generally regarded as poor programming practice. If a function reaches the end without encountering a `return` statement, control simply reverts back to the calling portion of the program without returning any information. The presence of an empty `return` statement (without the accompanying expression) is recommended in such situations, to clarify the logic and to accommodate future modifications to the function.

EXAMPLE 7.4 The following function accepts two integer quantities and determines the larger value, which is then displayed. The function does not return any information to the calling program.

```
maximum(int x, int y)        /* determine the larger of two integer quantities */
{
    int z;

    z = (x >= y) ? x : y;
    printf("\n\nMaximum value = %d", z);
    return;
}
```

Notice that an empty `return` statement is included, as a matter of good programming practice. The function would still work properly, however, if the `return` statement were not present.

EXAMPLE 7.5 The *factorial* of a positive integer quantity, n, is defined as $n! = 1 \times 2 \times 3 \times \cdots \times n$. Thus, $2! = 1 \times 2 = 2$; $3! = 1 \times 2 \times 3 = 6$; $4! = 1 \times 2 \times 3 \times 4 = 24$; and so on.

The function shown below calculates the factorial of a given positive integer n. The factorial is returned as a long integer quantity, since factorials grow in magnitude very rapidly as n increases. (For example, 8! = 40,320. This value, expressed as an ordinary integer, may be too large for some computers.)

```
long int factorial(int n)        /* calculate the factorial of n */

{
    int i;
    long int prod = 1;

    if (n > 1)
       for (i = 2; i <= n; ++i)
           prod *= i;
    return(prod);
}
```

Notice the `long int` type specification that is included in the first line of the function definition. The local variable `prod` is declared to be a long integer within the function. It is assigned an initial value of 1, though its value is recalculated within a `for` loop. The final value of `prod`, which is returned by the function, represents the desired value of n factorial.

If the data type specified in the first line is inconsistent with the expression appearing in the `return` statement, the compiler will attempt to convert the quantity represented by the expression to the data type specified in the first line. This could result in a compilation error, or it may involve a partial loss of data (e.g., due to truncation). In any event, inconsistencies of this type should be avoided.

EXAMPLE 7.6 The following function definition is identical to that in Example 7.5 except that the first line does not include a type specification for the value that is returned by the function.

```
factorial(int n)        /* calculate the factorial of n */

{
    int i;
    long int prod = 1;

    if (n > 1)
       for (i = 2; i <= n; ++i)
           prod *= i;
    return(prod);
}
```

The function expects to return an ordinary integer quantity, since there is no explicit type declaration in the first line of the function definition. However the quantity being returned (`prod`) is declared as a long integer within the function. This inconsistency can result in an error. (Some compilers will generate a diagnostic error and then stop without completing the compilation.) The problem can be avoided, however, by adding a `long int` type declaration to the first line of the function definition, as in Example 7.5.

The keyword `void` can be used as a type specifier when defining a function that does not return anything, or when the function definition does not include any arguments. The presence of this keyword is not mandatory, but it is good programming practice to make use of this feature.

EXAMPLE 7.7 Consider once again the function presented in Example 7.4, which accepts two integer quantities and displays the larger of the two. Recall that this function does not return anything to the calling portion of the program. Therefore, the function can be written as

```
void maximum(x, y)        /* determine the larger of two integer quantities */
int x, y;
{
    int z;

    z = (x >= y) ? x : y;
    printf("\n\nMaximum value = %d", z);
    return;
}
```

This function is identical to that shown in Example 7.4 except that the keyword `void` has been added to the first line, indicating that the function does not return anything.

7.3 ACCESSING A FUNCTION

A function can be *accessed* (i.e., *called*) by specifying its name, followed by a list of arguments enclosed in parentheses and separated by commas. If the function call does not require any arguments, an empty pair of parentheses must follow the name of the function. The function call may be a part of a simple expression (such as an assignment statement), or it may be one of the operands within a more complex expression.

The arguments appearing in the function call are referred to as *actual arguments*, in contrast to the formal arguments that appear in the first line of the function definition. (They are also known simply as *arguments*, or as *actual parameters*.) In a normal function call, there will be one actual argument for each formal argument. The actual arguments may be expressed as constants, single variables, or more complex expressions. However, each actual argument must be of the same data type as its corresponding formal argument. Remember that it is the *value* of each actual argument that is transferred into the function and assigned to the corresponding formal argument.

If the function returns a value, the function access is often written as an assignment statement; e.g.,

```
y = polynomial(x);
```

This function access causes the value returned by the function to be assigned to the variable `y`.

On the other hand, if the function does not return anything, the function access appears by itself; e.g.,

```
display(a, b, c);
```

This function access causes the values of `a`, `b` and `c` to be processed internally (i.e., displayed) within the function.

EXAMPLE 7.8 Consider once again the program originally shown in Example 7.1, which reads in a single lowercase character, converts it to uppercase using a programmer-defined function, and then displays the uppercase equivalent.

```
/* convert a lowercase character to uppercase using a programmer-defined function */
#include <stdio.h>

char lower_to_upper(char c1)          /* function definition */
{
    char c2;

    c2 = (c1 >= 'a' && c1 <= 'z') ? ('A' + c1 - 'a') : c1;
    return(c2);
}
```

```
void main(void)

{
    char lower, upper;

    printf("Please enter a lowercase character: ");
    scanf("%c", &lower);
    upper = lower_to_upper(lower);
    printf("\nThe uppercase equivalent is %c\n\n", upper);
}
```

Within this program, `main` contains only one call to the programmer-defined function `lower_to_upper`. The call is a part of the assignment expression `upper = lower_to_upper(lower)`.

The function call contains one actual argument, the char-type variable `lower`. Note that the corresponding formal argument, `c1`, within the function definition is also a char-type variable.

When the function is accessed, the value of `lower` to be transferred to the function. This value is represented by `c1` within the function. The value of the uppercase equivalent, `c2`, is then determined and returned to the calling portion of the program, where it is assigned to the char-type variable `upper`.

The last two statements in `main` can be combined to read

```
    printf("\nThe uppercase equivalent is %c\n\n", lower_to_upper(lower));
```

The call to `lower_to_upper` is now an actual argument for the library function `printf`. Also, note that the variable `upper` is no longer required.

Finally, notice the manner in which the first line of `main` is written, i.e., `void main(void)`. This is permitted under the ANSI standard, though some compilers do not accept the `void` return type. Hence, many authors (and many programmers) write the first line of `main` as `main(void)`, or simply `main()`. We will follow the latter designation throughout the remainder of this book.

There may be several different calls to the same function from various places within a program. The actual arguments may differ from one function call to another. Within each function call, however, *the actual arguments must correspond to the formal arguments in the function definition; i.e., the number of actual arguments must be the same as the number of formal arguments, and each actual argument must be of the same data type as its corresponding formal argument.*

EXAMPLE 7.9 Largest of Three Integer Quantities The following program determines the largest of three integer quantities. This program makes use of a function that determines the larger of two integer quantities. The function is similar to that defined in Example 7.4, except that the present function returns the larger value to the calling program rather than displaying it.

The overall strategy is to determine the larger of the first two quantities, and then compare this value with the third quantity. The largest quantity is then displayed by the main part of the program.

```
/* determine the largest of three integer quantities */

#include <stdio.h>


int maximum(int x, int y)        /* determine the larger of two integer quantities */

{
    int z;

    z = (x >= y) ? x : y;
    return(z);
}
```

```
main()
{
    int a, b, c, d;

    /* read the integer quantities */
    printf("\na = ");
    scanf("%d", &a);
    printf("\nb = ");
    scanf("%d", &b);
    printf("\nc = ");
    scanf("%d", &c);

    /* calculate and display the maximum value */

    d = maximum(a, b);
    printf("\n\nmaximum = %d", maximum(c, d));
}
```

The function `maximum` is accessed from two different places in `main`. In the first call to `maximum` the actual arguments are the variables `a` and `b`, whereas the arguments are `c` and `d` in the second call (`d` is a temporary variable representing the maximum value of `a` and `b`).

Note the two statements in `main` that access `maximum`, i.e.,

```
d = maximum(a, b);
printf("\n\nmaximum = %d", maximum(c, d));
```

These two statements can be replaced by a single statement; e.g.,

```
printf("\n\nmaximum = %d", maximum(c, maximum(a, b)));
```

In this statement we see that one of the calls to `maximum` is an argument for the other call. Thus the calls are embedded, one within the other, and the intermediary variable, `d`, is not required. Such embedded function calls are permissible, though their logic may be unclear. Hence, they should generally be avoided by beginning programmers.

7.4 FUNCTION PROTOTYPES

In the programs that we have examined earlier in this chapter, the programmer-defined function has always *preceded* `main`. Thus, when these programs are compiled, the programmer-defined function will have been defined before the first function access. However, many programmers prefer a "top-down" approach, in which `main` appears ahead of the programmer-defined function definition. In such situations the function access (within `main`) will precede the function definition. This can be confusing to the compiler, unless the compiler is first alerted to the fact that the function being accessed will be defined later in the program. A *function prototype* is used for this purpose.

Function prototypes are usually written at the beginning of a program, ahead of any programmer-defined functions (including `main`). The general form of a function prototype is

data-type name(*type 1 arg 1, type 2 arg 2, . . ., type n arg n*);

where *data-type* represents the data type of the item that is returned by the function, *name* represents the function name, and *type 1*, *type 2*, . . . , *type n* represent the data types of the arguments *arg 1*, *arg 2*, . . . , *arg n*. Notice that a function prototype resembles the first line of a function definition (though a function prototype ends with a semicolon).

The names of the arguments within the function prototype need not be declared elsewhere in the program, since these are "dummy" argument names that are recognized only within the prototype. In fact, the argument names can be omitted (though it is not a good idea to do so); however, the argument *data types* are essential.

In practice, the argument names are usually included and are often the same as the names of the actual arguments appearing in one of the function calls. The data types of the actual arguments must conform to the data types of the arguments within the prototype.

Function prototypes are not mandatory in C. They are desirable, however, because they further facilitate error checking between the calls to a function and the corresponding function definition.

EXAMPLE 7.10 Calculating Factorials Here is a complete program to calculate the factorial of a positive integer quantity. The program utilizes the function `factorial`, defined in Example 7.5. Note that the function definition precedes `main`, as in the earlier programming examples within this chapter.

```
/* calculate the factorial of an integer quantity */

#include <stdio.h>

long int factorial(int n)

/* calculate the factorial of n */

{
    int i;
    long int prod = 1;

    if (n > 1)
       for (i = 2; i <= n; ++i)
           prod *= i;
    return(prod);
}


main()

{
    int n;

    /* read in the integer quantity */

    printf("\nn = ");
    scanf("%d", &n);

    /* calculate and display the factorial */

    printf("\n%d! = %ld", n, factorial(n));
}
```

The programmer-defined function (`factorial`) makes use of an integer argument (`n`) and two local variables—an ordinary integer (`i`) and a long integer (`prod`). Since the function returns a long integer, the type declaration `long int` appears in the first line of the function definition.

Here is another version of the program, written top-down (i.e, with `main` appearing ahead of `factorial`). Notice the presence of the function prototype at the beginning of the program. The function prototype indicates that a function called `factorial`, which accepts an integer quantity and returns a long integer quantity, will be defined later in the program.

```
/* calculate the factorial of an integer quantity */

#include <stdio.h>

long int factorial(int n);      /* function prototype */
```

```
main()
{
    int n;

    /* read in the integer quantity */

    printf("\nn = ");
    scanf("%d", &n);

    /* calculate and display the factorial */

    printf("\n%d! = %ld", factorial(n));
}

long int factorial(int n)

/* calculate the factorial of n */

{
    int i;
    long int prod = 1;

    if (n > 1)
       for (i = 2; i <= n; ++i)
           prod *= i;
    return(prod);
}
```

Function calls can span several levels within a program. That is, function A can call function B, which can call function C, etc. Also, function A can call function C directly, and so on.

EXAMPLE 7.11 Simulation of a Game of Chance (Shooting Craps) Here is an interesting programming problem that includes multiple function calls at several different levels. Both library functions and programmer-defined functions are required.

Craps is a popular dice game in which you throw a pair of dice one or more times until you either win or lose. The game can be simulated on a computer by generating random numbers rather than actually throwing the dice.

There are two ways to win in craps. You can throw the dice once and obtain a score of either 7 or 11; or you can obtain a 4, 5, 6, 8, 9 or 10 on the first throw and then repeat the same score on a subsequent throw before obtaining a 7. Conversely, there are two ways to lose. You can throw the dice once and obtain a 2, 3 or 12; or you can obtain a 4, 5, 6, 8, 9 or 10 on the first throw and then obtain a 7 on a subsequent throw before repeating your original score.

We will develop the game interactively, so that one throw of the dice will be simulated each time you press the Enter key. A message will then appear indicating the outcome of each throw. At the end of each game, you will be asked whether or not you want to continue to play.

Our program will require a random number generator that produces uniformly distributed integers between 1 and 6. (By *uniformly distributed* we mean that any integer between 1 and 6 is just as likely to appear as any other integer.) Most versions of C include a random number generator in their library routines. These random number generators typically return a floating-point number that is uniformly distributed between 0 and 1, or an integer quantity that is uniformly distributed between 0 and some very large integer value.

We will employ a random number generation routine called `rand`, which returns a uniformly distributed integer between 0 and $2^{15} - 1$ (i.e., between 0 and 32,767). We then convert each random integer quantity to a floating-point number, `x`, which varies from 0 to 0.99999$\cdots$. To do so, we write

```
x = rand() / 32768.0
```

Note that the denominator is written as a floating-point constant. This forces the quotient, and hence `x`, to be a floating-point quantity.

The expression

```
(int) (6 * x)
```

will result in a truncated integer whose value will be uniformly distributed between 0 and 5. Thus, we obtain the desired value simply by adding 1; i.e.,

```
n = 1 + (int) (6 * x)
```

This value will represent the random outcome of rolling one die. If we repeat this process a second time and add the results, we obtain the result of rolling two dice.

The following function utilizes the above strategy to simulate one throw of a pair of dice.

```
int throw(void)   /* simulate one throw of a pair of dice */
{
    float x1, x2;  /* random floating-point numbers between 0 and 1 */
    int n1, n2;    /* random integers between 1 and 6 */

    x1 = rand() / 32768.0;
    x2 = rand() / 32768.0;

    n1 = 1 + (int) (6 * x1);   /* simulate first die */
    n2 = 1 + (int) (6 * x2);   /* simulate second die */

    return(n1 + n2);           /* score is sum of two dice */
}
```

The function returns the result of each throw (an integer quantity whose value varies between 2 and 12). Note that this final result will *not* be uniformly distributed, even though the individual values of `n1` and `n2` are.

Now let us define another function, called `play`, which can simulate one complete game of craps. Thus, the dice will be thrown as many times as is necessary to establish either a win or a loss. This function will therefore access `throw`. The complete rules of craps will also be built into this function.

In pseudocode, we can write the function `play` as

```
void play(void)   /* simulate one complete game */
{
    int score1, score2;

    /* instruct the user to throw the dice */

    /* initialize the random number generator */

    score1 = throw();

    switch (score1)   {

    case 7:
    case 11:

         /* display a message indicating a win on the first throw */

    case 2:
    case 3:
    case 12:

         /* display a message indicating a loss on the first throw */
```

```
        case 4:
        case 5:
        case 6:
        case 8:
        case 9:
        case 10:
             do   {
                      /* instruct the user to throw the dice again */
                      score2 = throw();
             }   while (score2 != score1 && score2 != 7);

             if (score2 == score1)
                /* display a message indicating a win */
             else
                /* display a message indicating a loss */
        }
        return;
}
```

The main routine will control the execution of the game. This routine will consist of a `while` loop containing some interactive input/output and a call to `play`. Thus, we can write the pseudocode for `main` as

```
main()
{
    /* declarations */
    /* initialize the random number generator */
    /* generate a welcoming message */
    while ( /* player wants to continue */ )   {
       play();
       /* ask if player wants to continue */
    }
    /*  generate a sign-off message */
}
```

The library function `srand` will be used to initialize the random number generator. This function requires a positive integer, called a *seed*, which establishes the sequence of random numbers generated by `rand`. A different sequence will be generated for each seed. For convenience, we can include a value for the seed as a symbolic constant within the program. (If the program is executed repeatedly with the same seed, the same sequence of random numbers will be generated each time. This is helpful when debugging the program.)

Here is the complete C program, written top-down.

```
/* simulation of a craps game */

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

#define  SEED   12345
```

```
void play(void);                /* function prototype */
int throw(void);                /* function prototype */

main()

{
    char answer = 'Y';

    printf("Welcome to the Game of CRAPS\n\n");
    printf("To throw the dice, press Enter\n\n");

    srand(SEED);   /* initialize the random number generator */

    /* main loop */

    while (toupper(answer) != 'N')   {
        play();
        printf("\nDo you want to play again? (Y/N) ");
        scanf(" %c", &answer);
        printf("\n");
    }
    printf("Bye, have a nice day");
}

void play(void)   /* simulate one complete game */

{
    int score1, score2;
    char dummy;

    printf("\nPlease throw the dice . . .");
    scanf("%c", &dummy);
    printf("\n");
    score1 = throw();
    printf("\n%2d", score1);

    switch (score1)  {

    case 7:   /* win on first throw */
    case 11:

         printf(" - Congratulations!  You WIN on the first throw\n");
         break;

    case 2:   /* lose on first throw */
    case 3:
    case 12:

         printf(" - Sorry, you LOSE on the first throw\n");
         break;

    case 4:   /* additional throws are required */
    case 5:
    case 6:
    case 8:
    case 9:
    case 10:
```

```
        do   {
                printf(" - Throw the dice again . . .");
                scanf("%c", &dummy);
                score2 = throw();
                printf("\n%2d", score2);
        }  while (score2 != score1 && score2 != 7);

        if (score2 == score1)
           printf(" - You WIN by matching your first score\n");
        else
           printf(" - You LOSE by failing to match your first score\n");
        break;
    }

    return;
}

int throw(void)   /* simulate one throw of a pair of dice */

{
    float x1, x2;  /* random floating-point numbers between 0 and 1 */
    int n1, n2;    /* random integers between 1 and 6 */

    x1 = rand() / 32768.0;
    x2 = rand() / 32768.0;

    n1 = 1 + (int) (6 * x1);   /* simulate first die */
    n2 = 1 + (int) (6 * x2);   /* simulate second die */

    return(n1 + n2);           /* score is sum of two dice */
}
```

Notice that `main` calls `srand` and `play`. One argument is passed to `srand` (the value of the seed), but no arguments are passed to `play`. Also, note that `play` calls `throw` from two different places, and `throw` calls `rand` from two different places. There are no arguments passed from `play` to `throw` or from `throw` to `rand`. However, `rand` returns a random integer to `throw`, and `throw` returns the value of an integer expression (the outcome of one throw of the dice) to `play`. Notice that `play` does not return any information to `main`.

Within `play`, there are two references to the `scanf` function, each of which enters a value for the variable `dummy`. It should be understood that `dummy` is not actually used within the program. The `scanf` functions are present simply to halt the program temporarily, until the user presses the `Enter` key (to simulate a new throw of the dice).

This program is designed to run in an interactive environment, such as on a personal computer. A typical set of output is shown below. The user's responses are underlined for clarity.

```
Welcome to the Game of CRAPS

To throw the dice, press Enter (Enter)


Please throw the dice . . .

 6 - Throw the dice again . . .

10 - Throw the dice again . . .

 7 - You LOSE by failing to match your first score
```

```
Do you want to play again? (Y/N) y

Please throw the dice . . .
 7 - Congratulations!  You WIN on the first throw
Do you want to play again? (Y/N) y

Please throw the dice . . .
11 - Congratulations!  You WIN on the first throw
Do you want to play again? (Y/N) y

Please throw the dice . . .
 8 - Throw the dice again . . .
 5 - Throw the dice again . . .
 7 - You LOSE by failing to match your first score
Do you want to play again? (Y/N) y

Please throw the dice . . .
 6 - Throw the dice again . . .
 4 - Throw the dice again . . .
 6 - You WIN by matching your first score
Do you want to play again? (Y/N) y

Please throw the dice . . .
 3 - Sorry, you LOSE on the first throw
Do you want to play again? (Y/N) n
Bye, have a nice day
```

7.5 PASSING ARGUMENTS TO A FUNCTION

When a single value is passed to a function via an actual argument, the value of the actual argument is *copied* into the function. Therefore, *the value of the corresponding formal argument can be altered within the function, but the value of the actual argument within the calling routine will not change.* This procedure for passing the value of an argument to a function is known as *passing by value.*

EXAMPLE 7.12 Here is a simple C program containing a function that alters the value of its argument.

```
#include <stdio.h>

void modify(int a);      /* function prototype */

main()
{
    int a = 2;

    printf("\na = %d  (from main, before calling the function)", a);
    modify(a);
    printf("\n\na = %d  (from main, after calling the function)", a);
}

void modify(int a)
{
    a *= 3;
    printf("\n\na = %d  (from the function, after being modified)", a);
    return;
}
```

The original value of a (i.e., a = 2) is displayed when `main` begins execution. This value is then passed to the function `modify`, where it is multiplied by 3 and the new value displayed. Note that it is the *altered* value of the formal argument that is displayed within the function. Finally, the value of a within `main` (i.e., the actual argument) is again displayed, after control is transferred back to `main` from `modify`.

When the program is executed, the following output is generated.

```
a = 2  (from main, before calling the function)

a = 6  (from the function, after being modified)

a = 2  (from main, after calling the function)
```

These results show that a is *not* altered within `main`, even though the corresponding value of a *is* changed within `modify`.

Passing an argument by value has advantages and disadvantages. On the plus side, it allows a single-valued actual argument to be written as an expression rather than being restricted to a single variable. Moreover, if the actual argument is expressed simply as a single variable, it protects the value of this variable from alterations within the function. On the other hand, it does not allow information to be transferred back to the calling portion of the program via arguments. Thus, *passing by value is restricted to a one-way transfer of information.*

EXAMPLE 7.13 Calculating Depreciation Let us consider a variation of the depreciation program presented in Example 6.26. The overall objective is to calculate depreciation as a function of time using any one of three different commonly used methods, as before. Now, however, we will rewrite the program so that a separate function is used for each method. This approach offers us a cleaner way to organize the program into its logical components. In addition, we will move a block of repeated output instructions into a separate function, thus eliminating some redundant programming from the original version of the program.

We will also expand the generality of the program somewhat, by permitting different sets of depreciation calculations to be carried out on the same input data. Thus, at the end of each set of calculations the user will be asked if another set of calculations is desired. If the answer is yes, then the user will be asked whether or not to enter new data.

Here is the new version of the program, written top-down.

```
/* calculate depreciation using one of three different methods */

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

void sl(float val, int n);                                          /* funct prototype */
void ddb(float val, int n);                                         /* funct prototype */
void syd(float val, int n);                                         /* funct prototype */
void writeoutput(int year, float depreciation, float value);        /* funct prototype */

main()

{
    int n, choice = 0;
    float val;
    char answer1 = 'Y', answer2 = 'Y';

    while (toupper(answer1) != 'N')   {

          /* read input data */

          if (toupper(answer2) != 'N')   {
             printf("\nOriginal value: ");
             scanf("%f", &val);
             printf("Number of years: ");
             scanf("%d", &n);
          }
          printf("\nMethod: (1-SL  2-DDB  3-SYD) ");
          scanf("%d", &choice);

          switch (choice)  {

          case 1:     /* straight-line method */

               printf("\nStraight-Line Method\n\n");
               sl(val, n);
               break;

          case 2:     /* double-declining-balance method */

               printf("\nDouble-Declining-Balance Method\n\n");
               ddb(val, n);
               break;

          case 3:     /* sum-of-the-years'-digits method */

               printf("\nSum-Of-The-Years\'-Digits Method\n\n");
               syd(val, n);
          }

          printf("\n\nAnother calculation? (Y/N) ");
          scanf("%1s", &answer1);
          if (toupper(answer1) != 'N')   {
             printf("Enter a new set of data? (Y/N) ");
             scanf("%1s", &answer2);
          }
    }

    printf("\nGoodbye, have a nice day!\n");

}
```

```
void sl(float val, int n)    /* straight-line method */

{
    float deprec;
    int year;

    deprec = val/n;
    for (year = 1; year <= n; ++year)   {
        val -= deprec;
        writeoutput(year, deprec, val);
    }
    return;
}

void ddb(float val, int n)    /* double-declining-balance method */

{
    float deprec;
    int year;

    for (year = 1; year <= n; ++year)  {
        deprec = 2*val/n;
        val -= deprec;
        writeoutput(year, deprec, val);
    }
    return;
}

void syd(float val, int n)    /* sum-of-the-years'-digits method */

{
    float tag, deprec;
    int year;

    tag = val;
    for (year = 1; year <= n; ++year)   {
        deprec = (n-year+1)*tag / (n*(n+1)/2);
        val -= deprec;
        writeoutput(year, deprec, val);
    }
    return;
}

void writeoutput(int year, float depreciation, float value)    /* display output data */

{
    printf("End of Year %2d", year);
    printf("  Depreciation: %7.2f", depreciation);
    printf("  Current Value: %8.2f\n", value);
    return;
}
```

Notice that the `switch` statement is still employed, as in Example 6.26, though there are now only three choices rather than four. (The fourth choice, which ended the computation in the previous version, is now handled through interactive dialog at the end of each set of calculations.) A separate function is now provided for each type of calculation. In particular, the straight-line calculations are carried out within function `sl`, the double-declining-balance calculations within `ddb`, and the sum-of-the-years'-digits calculations within `syd`. Each of these functions includes the formal

arguments `val` and `n`, which represent the original value of the item and its lifetime, respectively. Note that the value of `val` is altered within each function, although the original value assigned to `val` remains unaltered within `main`. It is this feature that allows repeated sets of calculations with the same input data.

The last function, `writeoutput`, causes the results of each set of calculations to be displayed on a year-by-year basis. This function is accessed from `sl`, `ddb` and `syd`. In each call to `writeoutput`, the *altered* value of `val` is transferred as an actual argument, along with the current year (`year`) and the current year's depreciation (`deprec`). Note that these quantities are called `value`, `year` and `depreciation`, respectively, within `writeoutput`.

A sample interactive session which makes use of this program is shown below.

```
Original value: 8000
Number of years: 10

Method: (1-SL  2-DDB  3-SYD) 1

Straight-Line Method

End of Year  1  Depreciation:  800.00  Current Value:  7200.00
End of Year  2  Depreciation:  800.00  Current Value:  6400.00
End of Year  3  Depreciation:  800.00  Current Value:  5600.00
End of Year  4  Depreciation:  800.00  Current Value:  4800.00
End of Year  5  Depreciation:  800.00  Current Value:  4000.00
End of Year  6  Depreciation:  800.00  Current Value:  3200.00
End of Year  7  Depreciation:  800.00  Current Value:  2400.00
End of Year  8  Depreciation:  800.00  Current Value:  1600.00
End of Year  9  Depreciation:  800.00  Current Value:   800.00
End of Year 10  Depreciation:  800.00  Current Value:     0.00


Another calculation? (Y/N) y
Enter a new set of data? (Y/N) n

Method: (1-SL  2-DDB  3-SYD) 2

Double-Declining-Balance Method

End of Year  1  Depreciation: 1600.00  Current Value:  6400.00
End of Year  2  Depreciation: 1280.00  Current Value:  5120.00
End of Year  3  Depreciation: 1024.00  Current Value:  4096.00
End of Year  4  Depreciation:  819.20  Current Value:  3276.80
End of Year  5  Depreciation:  655.36  Current Value:  2621.44
End of Year  6  Depreciation:  524.29  Current Value:  2097.15
End of Year  7  Depreciation:  419.43  Current Value:  1677.72
End of Year  8  Depreciation:  335.54  Current Value:  1342.18
End of Year  9  Depreciation:  268.44  Current Value:  1073.74
End of Year 10  Depreciation:  214.75  Current Value:   858.99


Another calculation? (Y/N) y
Enter a new set of data? (Y/N) n

Method: (1-SL  2-DDB  3-SYD) 3

Sum-Of-The-Years'-Digits Method
```

```
End of Year  1  Depreciation: 1454.55  Current Value:  6545.45
End of Year  2  Depreciation: 1309.09  Current Value:  5236.36
End of Year  3  Depreciation: 1163.64  Current Value:  4072.73
End of Year  4  Depreciation: 1018.18  Current Value:  3054.55
End of Year  5  Depreciation:  872.73  Current Value:  2181.82
End of Year  6  Depreciation:  727.27  Current Value:  1454.55
End of Year  7  Depreciation:  581.82  Current Value:   872.73
End of Year  8  Depreciation:  436.36  Current Value:   436.36
End of Year  9  Depreciation:  290.91  Current Value:   145.45
End of Year 10  Depreciation:  145.45  Current Value:     0.00


Another calculation? (Y/N) y
Enter a new set of data? (Y/N) y

Original value: 5000
Number of years: 4

Method: (1-SL  2-DDB  3-SYD) 1

Straight-Line Method

End of Year  1  Depreciation: 1250.00  Current Value:  3750.00
End of Year  2  Depreciation: 1250.00  Current Value:  2500.00
End of Year  3  Depreciation: 1250.00  Current Value:  1250.00
End of Year  4  Depreciation: 1250.00  Current Value:     0.00


Another calculation? (Y/N) y
Enter a new set of data? (Y/N) n

Method: (1-SL  2-DDB  3-SYD) 2

Double-Declining-Balance Method

End of Year  1  Depreciation: 2500.00  Current Value:  2500.00
End of Year  2  Depreciation: 1250.00  Current Value:  1250.00
End of Year  3  Depreciation:  625.00  Current Value:   625.00
End of Year  4  Depreciation:  312.50  Current Value:   312.50


Another calculation? (Y/N) n

Goodbye, have a nice day!
```

Notice that two different sets of input data are processed. Depreciation is calculated for the first set using all three methods, and for the second set using only the first two methods. Thus, it is not necessary to reenter the input data simply to recalculate the depreciation using a different method.

Array arguments are passed differently than single-valued data items. If an array name is specified as an actual argument, the individual array elements are not copied. Instead, the *location* of the array (i.e., the location of the first element) is passed to the function. If an element of the array is then accessed within the function, the access will refer to the location of that array element relative to the location of the first element.

Thus, *any alteration to an array element within the function will carry over to the calling routine.* We will discuss this in greater detail in Chap. 9, when we formally consider arrays.

There are also other kinds of data structures that can be passed as arguments to a function. We will discuss the transfer of such arguments in later chapters, as the additional data structures are introduced.

7.6 RECURSION

Recursion is a process by which a function calls itself repeatedly, until some specified condition has been satisfied. The process is used for repetitive computations in which each action is stated in terms of a previous result. Many iterative (i.e., repetitive) problems can be written in this form.

In order to solve a problem recursively, two conditions must be satisfied. First, the problem must be written in a recursive form, and second, the problem statement must include a stopping condition. Suppose, for example, we wish to calculate the factorial of a positive integer quantity. We would normally express this problem as $n! = 1 \times 2 \times 3 \times \cdots \times n$, where n is the specified positive integer (see Example 7.5). However, we can also express this problem in another way, by writing $n! = n \times (n - 1)!$ This is a recursive statement of the problem, in which the desired action (the calculation of $n!$) is expressed in terms of a previous result [the value of $(n - 1)!$, which is assumed to be known]. Also, we know that $1! = 1$ by definition. This last expression provides a stopping condition for the recursion.

EXAMPLE 7.14 Calculating Factorials In Example 7.10 we saw two versions of a program that calculates the factorial of a given input quantity, using a nonrecursive function to perform the actual calculations. Here is a program that carries out this same calculation using recursion.

```
/* calculate the factorial of an integer quantity using recursion */

#include <stdio.h>

long int factorial(int n);        /* function prototype */

main()

{
    int n;
    long int factorial(int n);

    /* read in the integer quantity */

    printf("n = ");
    scanf("%d", &n);

    /* calculate and display the factorial */

    printf("n! = %ld\n",  factorial(n));
}

long int factorial(int n)        /* calculate the factorial */

{
    if (n <= 1)
       return(1);
    else
       return(n * factorial(n - 1));
}
```

The main portion of the program simply reads the integer quantity n and then calls the long-integer recursive function `factorial`. (Recall that we use long integers for this calculation because factorials are such large integer quantities, even

for modest values of n.) The function `factorial` calls itself recursively, with an actual argument (`n - 1`) that decreases in magnitude for each successive call. The recursive calls terminate when the value of the actual argument becomes equal to 1.

Notice that the present form of `factorial` is simpler than the function presented in Example 7.10. The close correspondence between this function and the original problem definition, in recursive terms, should be readily apparent. In particular, note that the `if - else` statement includes a termination condition that becomes active when the value of n is less than or equal to 1. (Note that the value of n will never be less than 1 unless an improper initial value is entered into the computer.)

When the program is executed, the function `factorial` will be accessed repeatedly, once in `main` and $(n - 1)$ times within itself, though the person using the program will not be aware of this. Only the final answer will be displayed; for example,

```
n = 10

n! = 3628800
```

When a recursive program is executed, the recursive function calls are not executed immediately. Rather, they are placed on a *stack* until the condition that terminates the recursion is encountered.* The function calls are then executed in reverse order, as they are "popped" off the stack. Thus, when evaluating a factorial recursively, the function calls will proceed in the following order.

$n! = n \times (n - 1)!$

$(n - 1)! = (n - 1) \times (n - 2)!$

$(n - 2)! = (n - 2) \times (n - 3)!$

. . . . . .

$2! = 2 \times 1!$

The actual values will then be returned in the following reverse order.

$1! = 1$

$2! = 2 \times 1! = 2 \times 1 = 2$

$3! = 3 \times 2! = 3 \times 2 = 6$

$4! = 4 \times 3! = 4 \times 6 = 24$

. . . . . .

$n! = n \times (n - 1)! = \cdots$

This reversal in the order of execution is a characteristic of all functions that are executed recursively.

If a recursive function contains local variables, a *different* set of local variables will be created during each call. The names of the local variables will, of course, always be the same, as declared within the function. However, the variables will represent a different set of values each time the function is executed. Each set of values will be stored on the stack, so that they will be available as the recursive process "unwinds," i.e., as the various function calls are "popped" off the stack and executed.

EXAMPLE 7.15 Printing Backwards The following program reads in a line of text on a character-by-character basis, and then displays the characters in reverse order. The program utilizes recursion to carry out the reversal of the characters.

* A *stack* is a *last-in, first-out* data structure in which successive data items are "pushed down" upon preceding data items. The data items are later removed (i.e., they are "popped") from the stack in reverse order, as indicated by the last-in, first-out designation.

```
/* read a line of text and write it out backwards, using recursion */

#include <stdio.h>

#define EOLN  '\n'

void reverse(void);        /* function prototype */

main()

{
    printf("Please enter a line of text below\n");
    reverse();
}

void reverse(void)

/* read a line of characters and write it out backwards */

{
    char c;

    if ((c = getchar()) != EOLN) reverse();
    putchar(c);
    return;
}
```

The main portion of this program simply displays a prompt and then calls the function `reverse`, thus initiating the recursion. The recursive function `reverse` then proceeds to read single characters until an end-of-line designation (`\n`) is encountered. Each function call causes a new character (a new value for `c`) to be pushed onto the stack. Once the end of line is encountered, the successive characters are popped off the stack and displayed on a last-in, first-out basis. Thus, the characters are displayed in reverse order.

Suppose the program is executed with the following line of input:

```
Now is the time for all good men to come to the aid of their country!
```

Then the corresponding output will be

```
!yrtnuoc rieht fo dia eht ot emoc ot nem doog lla rof emit eht si woN
```

Sometimes a complicated repetitive process can be programmed very concisely using recursion, though the logic may be tricky. The following example provides a well-known illustration.

EXAMPLE 7.16 The Towers of Hanoi The *Towers of Hanoi* is a well-known children's game, played with three poles and a number of different-sized disks. Each disk has a hole in the center, allowing it to be stacked around any of the poles. Initially, the disks are stacked on the leftmost pole in the order of decreasing size, i.e., the largest on the bottom and the smallest on the top, as illustrated in Fig. 7.1.

The object of the game is to transfer the disks from the leftmost pole to the rightmost pole, without ever placing a larger disk on top of a smaller disk. Only one disk may be moved at a time, and each disk must always be placed around one of the poles.

The general strategy is to consider one of the poles to be the origin, and another to be the destination. The third pole will be used for intermediate storage, thus allowing the disks to be moved without placing a larger disk over a smaller one. Assume there are n disks, numbered from smallest to largest, as in Fig. 7.1. If the disks are initially stacked on the left pole, the problem of moving all n disks to the right pole can be stated in the following recursive manner.

1. Move the top $n - 1$ disks from the left pole to the center pole.

2. Move the nth disk (the largest disk) to the right pole.
3. Move the $n - 1$ disks on the center pole to the right pole.

The problem can be solved in this manner for any value of n greater than 0 ($n = 0$ represents a stopping condition).

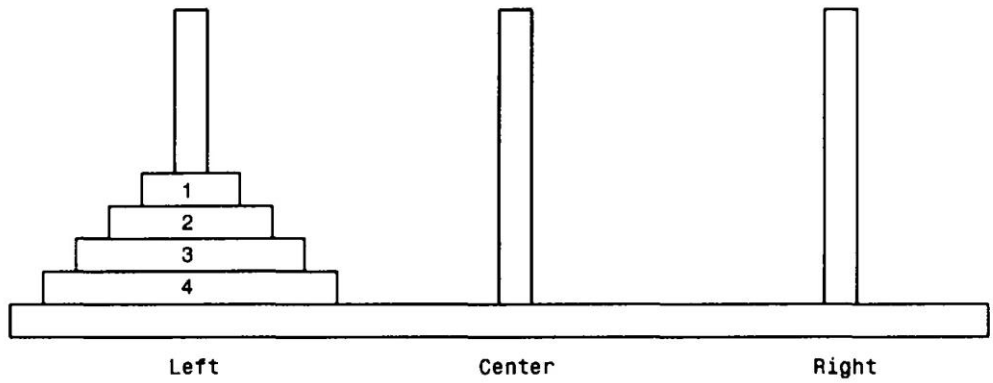

Fig. 7.1

In order to program this game we first label the poles so that the left pole is represented as `L`, the center pole as `C` and the right pole as `R`. We then construct a recursive function called `transfer` that will transfer n disks from one pole to another. Let us refer to the individual poles with the char-type variables `from`, `to` and `temp` for the origin, destination, and temporary storage, respectively. Thus, if we assign the character `L` to `from`, `R` to `to` and `C` to `temp`, we will in effect be specifying the movement of n disks from the leftmost pole to the rightmost pole, using the center pole for intermediate storage.

With this notation, the function will have the following skeletal structure.

```
void transfer(int n, char from, char to, char temp)
/*    n = number of disks
   from = origin
     to = destination
   temp = temporary storage */

{
   if (n > 0)   {

      /* move n-1 disks from their origin to the temporary pole */

      /* move the nth disk from its origin to its destination */

      /* move the n-1 disks from the temporary pole to their destination */

   }
}
```

The transfer of the $n - 1$ disks can be accomplished by a recursive call to `transfer`. Thus, we can write

```
transfer(n-1, from, temp, to);
```

for the first transfer, and

```
transfer(n-1, temp, to, from);
```

for the second. (*Note the order of the arguments in each call.*) The movement of the *n*th disk from the origin to the destination simply requires writing out the current values of `from` and `to`. Hence, the complete function can be written as follows.

```
void transfer(int n, char from, char to, char temp)

/* transfer n disks from one pole to another */

/* n    = number of disks
   from = origin
   to   = destination
   temp = temporary storage */

{
   if (n > 0)   {
      /* move n-1 disks from origin to temporary */
      transfer(n-1, from, temp, to);

      /* move nth disk from origin to destination */
      printf("Move disk %d from %c to %c\n", n, from, to);

      /* move n-1 disks from temporary to destination */
      transfer(n-1, temp, to, from);
   }
   return;
}
```

It is now a simple matter to add the main portion of the program, which merely reads in a value for *n* and then initiates the computation by calling `transfer`. In this first function call, the actual parameters will be specified as character constants, i.e.,

```
transfer(n, 'L', 'R', 'C');
```

This function call specifies the transfer of all *n* disks from the leftmost pole (the origin) to the rightmost pole (the destination), using the center pole for intermediate storage.

Here is the complete program.

```
/* the TOWERS OF HANOI - solved using recursion */

#include <stdio.h>

void transfer(int n, char from, char to, char temp);        /* function prototype */

main()
{
   int n;

   printf("Welcome to the TOWERS OF HANOI\n\n");
   printf("How many disks? ");
   scanf("%d", &n);
   printf("\n");
   transfer(n,'L','R','C');
}
```

```
void transfer(int n, char from, char to, char temp)

/* transfer n disks from one pole to another */

/* n    = number of disks
   from = origin
   to   = destination
   temp = temporary storage */

{
   if (n > 0)   {
      /* move n-1 disks from origin to temporary */
      transfer(n-1, from, temp, to);

      /* move nth disk from origin to destination */
      printf("Move disk %d from %c to %c\n", n, from, to);

      /* move n-1 disks from temporary to destination */
      transfer(n-1, temp, to, from);
   }
   return;
}
```

It should be understood that the function `transfer` receives a different set of values for its arguments each time the function is called. These sets of values will be pushed onto the stack independently of one another, and then popped from the stack at the proper time during the execution of the program. It is this ability to store and retrieve these independent sets of values that allows the recursion to work.

When the program is executed for the case where $n = 3$, the following output is obtained.

```
Welcome to the TOWERS OF HANOI

How many disks? 3

Move disk 1 from L to R
Move disk 2 from L to C
Move disk 1 from R to C
Move disk 3 from L to R
Move disk 1 from C to L
Move disk 2 from C to R
Move disk 1 from L to R
```

You should study these moves carefully to verify that the solution is indeed correct. The logic is very tricky, despite the apparent simplicity of the program.

We will see another programming example that utilizes recursion in Chap. 11, when we discuss linked lists.

The use of recursion is not necessarily the best way to approach a problem, even though the problem definition may be recursive in nature. A nonrecursive implementation may be more efficient, in terms of memory utilization and execution speed. Thus, the use of recursion may involve a tradeoff between simplicity and performance. Each problem should therefore be judged on its own individual merits.

Review Questions

7.1 What is a function? Are functions required when writing a C program?

7.2 State three advantages to the use of functions.

7.3 What is meant by a function call? From what parts of a program can a function be called?

7.4 What are arguments? What is their purpose? What other term is sometimes used for an argument?

7.5 What is the purpose of the `return` statement?

7.6 What are the two principal components of a function definition?

7.7 How is the first line of a function definition written? What is the purpose of each item, or group of items?

7.8 What are formal arguments? What are actual arguments? What is the relationship between formal arguments and actual arguments?

7.9 Describe some alternate terms that are used in place of *formal argument* and *actual argument*.

7.10 Can the names of the formal arguments within a function coincide with the names of other variables defined outside of the function? Explain.

7.11 Can the names of the formal arguments within a function coincide with the names of other variables defined within the function? Explain, and compare your answer with the answer to the last question.

7.12 Summarize the rules governing the use of the `return` statement. Can multiple expressions be included in a `return` statement? Can multiple `return` statements be included in a function?

7.13 What relationship must exist between the data type appearing at the beginning of the the first line of the function definition and the value returned by the `return` statement?

7.14 Why might a `return` statement be included in a function that does not return any value?

7.15 What is the purpose of the keyword `void`? Where is this keyword used?

7.16 Summarize the rules that apply to a function call. What relationships must be maintained between the actual arguments and the corresponding formal arguments in the function definition? Are the actual arguments subject to the same restrictions as the formal arguments?

7.17 Can a function be called from more than one place within a program?

7.18 What are function prototypes? What is their purpose? Where within a program are function prototypes normally placed?

7.19 Summarize the rules associated with function prototypes. What is the purpose of each item or group of items?

7.20 How are argument data types specified in a function prototype? What is the value of including argument data types in a function prototype?

7.21 When a function is accessed, must the names of the actual arguments agree with the names of the arguments in the corresponding function prototype?

7.22 Suppose function F1 calls function F2 within a C program. Does the order of the function definitions make any difference? Explain.

7.23 Describe the manner in which an actual argument passes information to a function. What name is associated with this process? What are the advantages and disadvantages to passing arguments in this manner?

7.24 What are differences between passing an array to a function and passing a single-valued data item to a function?

7.25 Suppose an array is passed to a function as an argument. If the value of an array element is altered within the function, will this change be recognized within the calling portion of the program?

7.26 What is recursion? What advantage is there in its use?

7.27 Explain why some problems can be solved either with or without recursion.

7.28 What is a stack? In what order is information added to and removed from a stack?

7.29 Explain what happens when a program containing recursive function calls is executed, in terms of information being added to and removed from the stack.

7.30 When a program containing recursive function calls is executed, how are the local variables within the recursive function interpreted?

7.31 If a repetitive process is programmed recursively, will the resulting program necessarily be more efficient than a nonrecursive version?

Problems

7.32 Explain the meaning of each of the following function prototypes.

(*a*)
```
int f(int a);
```
(*b*)
```
double f(double a, int b);
```
(*c*)
```
void f(long a, short b, unsigned c);
```
(*d*)
```
char f(void);
```
(*e*)
```
unsigned f(unsigned a, unsigned b);
```

7.33 Each of the following is the first line of a function definition. Explain the meaning of each.

(*a*)
```
float f(float a, float b)
```
(*b*)
```
long f(long a)
```
(*c*)
```
void f(int a)
```
(*d*)
```
char f(void)
```

7.34 Write an appropriate function call (function access) for each of the following functions.

(*a*)
```
float formula(float x)
{
    float y;

    y = 3 * x - 1;
    return(y);
}
```
(*b*)
```
void display(int a, int b)
{
    int c;

    c = sqrt(a * a + b * b);
    printf("c = %i\n", c);
}
```

7.35 Write the first line of the function definition, including the formal argument declarations, for each of the situations described below.

(*a*) A function called `sample` generates and returns an integer quantity.

(*b*) A function called `root` accepts two integer arguments and returns a floating-point result.

(*c*) A function called `convert` accepts a character and returns another character.

(*d*) A function called `transfer` accepts a long integer and returns a character.

(*e*) A function called `inverse` accepts a character and returns a long integer.

(*f*) A function called `process` accepts an integer and two floating-point quantities (in that order), and returns a double-precision quantity.

(*g*) A function called `value` accepts two double-precision quantities and a short-integer quantity (in that order). The input quantities are processed to yield a double-precision value which is displayed as a final result.

7.36 Write appropriate function prototypes for each of the skeletal outlines shown below.

(*a*)
```
main()
{
   int a, b, c;

   . . .

   c = funct1(a, b);

   . . .
}
```

```
int funct1(int x, int y)
{
   int z;

   z = . . .;
   return(z);
}
```

(*b*)

```
main()
{
   double a, b, c;

   . . .

   c = funct1(a, b);

   . . .
}

double funct1(double x, double y)
{
   double z;

   z = . . .;
   return(z);
}
```

(*c*)

```
main()
{
   int a;
   float b;
   long int c;

   . . .

   c = funct1(a, b);

   . . .
}

long int funct1(int x, float y)
{
   long int z;

   z = . . .;
   return(z);
}
```

(*d*)

```
main()
{
   double a, b, c, d;

   . . .

   c = funct1(a, b);

   . . .

   d = funct2(a + b, a + c);
}
```

```
double funct1(double x, double y)
{
   double z;

   . . .

   z = 10 * funct2(x, y);
   return(z);
}

double funct2(double x, double y)
{
   double z;

   z = . . .;
   return(z);
}
```

7.37 Describe the output generated by each of the following programs.

(*a*)
```
#include <stdio.h>

int funct(int count);

main()
{
   int a, count;

   for (count = 1; count <= 5; ++count)   {
       a = funct1(count);
       printf("%d  ", a);
   }
}

int funct1(int x)
{
   int y;

   y = x * x;
   return(y);
}
```

(*b*) Show how the preceding program can be written more concisely.

(*c*)
```
#include <stdio.h>

int funct1(int n);

main()
{
   int n = 10;

   printf("%d", funct1(n));
}

int funct1(int n)
{
   if (n > 0) return(n + funct1(n - 1));
}
```

(*d*)
```
#include <stdio.h>

int funct1(int n);

main()
{
   int n = 10;

   printf("%d", funct1(n));
}

int funct1(int n)
{
   if (n > 0) return(n + funct1(n - 2));
}
```

7.38 Express each of the following algebraic formulas in a recursive form.

(*a*) $y = (x_1 + x_2 + \cdots + x_n)$

(*b*) $y = 1 - x + x^2/2 - x^3/6 + x^4/24 + \cdots + (-1)^n x^n/n!$

(*c*) $p = (f_1 * f_2 * \cdots * f_t)$

Programming Problems

7.39 Write a function that will calculate and display the real roots of the quadratic equation

$$ax^2 + bx + c = 0$$

using the quadratic formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Assume that a, b and c are floating-point arguments whose values are given, and that x_1 and x_2 are floating-point variables. Also, assume that $b^2 > 4*a*c$, so that the calculated roots will always be real.

7.40 Write a complete C program that will calculate the real roots of the quadratic equation

$$ax^2 + bx + c = 0$$

using the quadratic formula, as described in the previous problem. Read the coefficients a, b and c in the main portion of the program. Then access the function written for the preceding problem in order to obtain the desired solution. Finally, display the values of the coefficients, followed by the calculated values of x_1 and x_2. Be sure that all of the output is clearly labeled.

Test the program using the following data:

a	b	c
2	6	1
3	3	0
1	3	1

7.41 Modify the function written for Prob. 7.39 so that *all* roots of the quadratic equation

$$ax^2 + bx + c = 0$$

will be calculated, given the values of a, b and c. Note that the roots will be repeated (i.e., there will only be one real root) if $b^2 = 4*a*c$. Also, the roots will be complex if $b^2 < 4*a*c$. In this case, the real part of each root will be determined as

$-b/(2*a)$

and the imaginary parts will be calculated as

$$\pm\left(\sqrt{4ac-b^2}\right)i$$

where i represents $\sqrt{-1}$

7.42 Modify the C program written for Prob. 7.40 so that *all* roots of the quadratic equation

$$ax^2 + bx + c = 0$$

will be calculated, using the function written for Prob. 7.41. Be sure that all of the output is clearly labeled. Test the program using the following data:

a	b	c
2	6	1
3	3	0
1	3	1
0	12	−3
3	6	3
2	−4	3

7.43 Write a function that will allow a floating-point number to be raised to an integer power. In other words, we wish to evaluate the formula

$$y = x^n$$

where y and x are floating-point variables and n is an integer variable.

7.44 Write a complete C program that will read in numerical values for x and n, evaluate the formula

$$y = x^n$$

using the function written for Prob. 7.43, and then display the calculated result. Test the program using the following data:

x	n	x	n
2	3	1.5	3
2	12	1.5	10
2	−5	1.5	−5
−3	3	0.2	3
−3	7	0.2	5
−3	−5	0.2	−5

7.45 Expand the function written for Prob. 7.43 so that positive values of x can be raised to *any* power, integer or floating-point. (*Hint*: Use the formula

$$y = x^n = e^{(n \ln x)}.$$

Remember to include a test for inappropriate values of x.)

Include this function in the program written for Prob. 7.44. Test the program using the data given in Prob. 7.44, and the following additional data.

x	n	x	n
2	0.2	1.5	0.2
2	−0.8	1.5	−0.8
−3	0.2	0.2	0.2
−3	−0.8	0.2	−0.8
		0.2	0.0

7.46 Modify the program for calculating the solution of an algebraic equation, given in Example 6.22, so that each iteration is carried out within a separate function. Compile and execute the program to be sure that it runs correctly.

7.47 Modify the program for averaging a list of numbers, given in Example 6.17, so that it makes use of a function to read in the numbers and return their sum. Test the program using the following 10 numbers:

27.5	87.0
13.4	39.9
53.8	47.7
29.2	8.1
74.5	63.2

7.48 Modify the program for carrying out compound interest calculations given in Example 5.2 so that the actual calculations are carried out in a programmer-defined function. Write the function so that the values of P, r and n are entered as arguments, and the calculated value of F is returned. Test the program using the following data.

P	r	n
1000	6	20
1000	6.25	20
333.33	8.75	20
333.33	8.75	22.5

7.49 For each of the following problems, write a complete C program that includes a recursive function.

(*a*) The *Legendre polynomials* can be calculated by means of the formulas $P_0 = 1$, $P_1 = x$,

$$P_n = [(2n - 1) / n] x P_{n-1} - [(n - 1) / n] P_{n-2}$$

where $n = 2, 3, 4, \ldots$ and x is any floating-point number between -1 and 1. (Note that the Legendre polynomials are floating-point quantities.)

Generate the first n Legendre polynomials. Let the values of n and x be input parameters.

(*b*) Determine the cumulative sum of n floating-point numbers [see Prob. 7.38(*a*)]. Read a new number into the computer during each call to the recursive function.

(*c*) Evaluate the first n terms in the series specified in Prob. 7.38(*b*). Enter n as an input parameter.

(*d*) Determine the cumulative product of n floating-point numbers [see Prob. 7.38(*c*)]. Read a new number into the computer during each call to the recursive function.

Additional programming problems involving the use of functions can be found at the end of Chap. 8.