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.

$$n! = n \times (n-1)!$$
$$(n-1)! = (n-1) \times (n-2)!$$
$$(n-2)! = (n-2) \times (n-3)!$$
$$\cdots\cdots\cdots\cdots\cdots\cdots$$
$$2! = 2 \times 1!$$

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

$$1! = 1$$
$$2! = 2 \times 1! = 2 \times 1 = 2$$
$$3! = 3 \times 2! = 3 \times 2 = 6$$
$$4! = 4 \times 3! = 4 \times 6 = 24$$
$$\cdots\cdots\cdots\cdots\cdots\cdots$$
$$n! = n \times (n-1)! = \cdots$$

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).

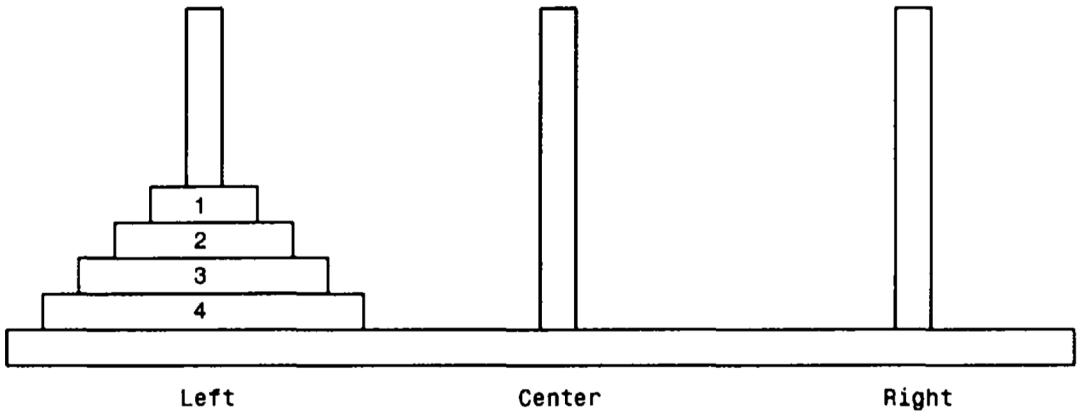

Fig. 7.1

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.