

Chapter 10

Pointers

A *pointer* is a variable that represents the *location* (rather than the *value*) of a data item, such as a variable or an array element. Pointers are used frequently in C, as they have a number of useful applications. For example, pointers can be used to pass information back and forth between a function and its reference point. In particular, pointers provide a way to return multiple data items from a function via function arguments. Pointers also permit references to other functions to be specified as arguments to a given function. This has the effect of passing functions as arguments to the given function.

Pointers are also closely associated with arrays and therefore provide an alternate way to access individual array elements. Moreover, pointers provide a convenient way to represent multidimensional arrays, allowing a single multidimensional array to be replaced by a lower-dimensional array of pointers. This feature permits a group of strings to be represented within a single array, though the individual strings may differ in length.

10.1 FUNDAMENTALS

Within the computer's memory, every stored data item occupies one or more contiguous memory cells (i.e., adjacent words or bytes). The number of memory cells required to store a data item depends on the type of data item. For example, a single character will typically be stored in one byte (8 bits) of memory; an integer usually requires two contiguous bytes; a floating-point number may require four contiguous bytes; and a double-precision quantity may require eight contiguous bytes. (See Chap. 2 and Appendix D.)

Suppose *v* is a variable that represents some particular data item. The compiler will automatically assign memory cells for this data item. The data item can then be accessed if we know the location (i.e., the *address*) of the first memory cell.* The address of *v*'s memory location can be determined by the expression `&v`, where `&` is a unary operator, called the *address operator*, that evaluates the address of its operand.

Now let us assign the address of *v* to another variable, *pv*. Thus,

$$pv = \&v$$

This new variable is called a *pointer* to *v*, since it "points" to the location where *v* is stored in memory. Remember, however, that *pv* represents *v*'s *address*, not its *value*. Thus, *pv* is referred to as a *pointer variable*. The relationship between *pv* and *v* is illustrated in Fig. 10.1.

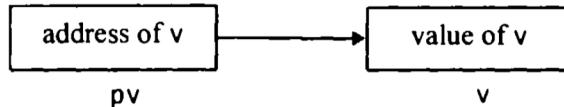


Fig. 10.1 Relationship between *pv* and *v* (where *pv* = `&v` and *v* = `*pv`)

The data item represented by *v* (i.e., the data item stored in *v*'s memory cells) can be accessed by the expression `*pv`, where `*` is a unary operator, called the *indirection operator*, that operates only on a pointer

* Adjacent memory cells within a computer are numbered consecutively, from the beginning to the end of the memory area. The number associated with each memory cell is known as the memory cell's *address*. Most computers use a hexadecimal numbering system to designate the addresses of consecutive memory cells, though some computers use an octal numbering system (see Appendix A).

variable. Therefore, `*pv` and `v` both represent the same data item (i.e., the contents of the same memory cells). Furthermore, if we write `pv = &v` and `u = *pv`, then `u` and `v` will both represent the same value; i.e., the value of `v` will indirectly be assigned to `u`. (It is assumed that `u` and `v` are of the same data type.)

EXAMPLE 10.1 Shown below is a simple program that illustrates the relationship between two integer variables, their corresponding addresses and their associated pointers.

```
#include <stdio.h>

main()
{
    int u = 3;
    int v;
    int *pu;      /* pointer to an integer */
    int *pv;      /* pointer to an integer */

    pu = &u;      /* assign address of u to pu */
    v = *pu;      /* assign value of u to v */
    pv = &v;      /* assign address of v to pv */

    printf("\nu=%d  &u=%X  pu=%X  *pu=%d", u, &u, pu, *pu);
    printf("\nv=%d  &v=%X  pv=%X  *pv=%d", v, &v, pv, *pv);
}
```

Note that `pu` is a pointer to `u`, and `pv` is a pointer to `v`. Therefore `pu` represents the address of `u`, and `pv` represents the address of `v`. (Pointer declarations will be discussed in the next section.)

Execution of this program results in the following output.

```
u=3  &u=F8E  pu=F8E  *pu=3
v=3  &v=F8C  pv=F8C  *pv=3
```

In the first line, we see that `u` represents the value 3, as specified in the declaration statement. The address of `u` is determined automatically by the compiler as F8E (hexadecimal). The pointer `pu` is assigned this value; hence, `pu` also represents the (hexadecimal) address F8E. Finally, the value to which `pu` points (i.e., the value stored in the memory cell whose address is F8E) is 3, as expected.

Similarly, the second line shows that `v` also represents the value 3. This is expected, since we have assigned the value `*pu` to `v`. The address of `v`, and hence the value of `pv`, is F8C. Notice that `u` and `v` have different addresses. And finally, we see that the value to which `pv` points is 3, as expected.

The relationships between `pu` and `u`, and `pv` and `v`, are shown in Fig. 10.2. Note that the memory locations of the pointer variables (i.e., address EC7 for `pu`, and EC5 for `pv`) are not displayed by the program.

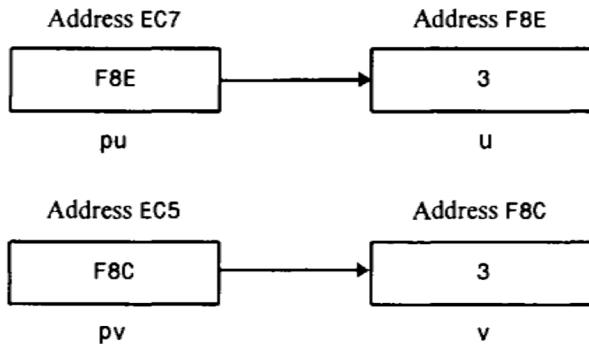


Fig. 10.2

The unary operators & and * are members of the same precedence group as the other unary operators, i.e., -, ++, --, !, sizeof and (type), which were presented in Chap. 3. Remember that this group of operators has a higher precedence than the groups containing the arithmetic operators, and that the associativity of the unary operators is right to left (see Appendix C).

The address operator (&) must act upon operands that are associated with unique addresses, such as ordinary variables or single array elements. Thus *the address operator cannot act upon arithmetic expressions*, such as $2 * (u + v)$.

The indirection operator (*) can only act upon operands that are pointers (e.g., pointer variables). However, if pv points to v (i.e., $pv = \&v$), then an expression such as $*pv$ can be used interchangeably with its corresponding variable v . Thus, an indirect reference (e.g., $*pv$) can appear in place of an ordinary variable (e.g., v) within a more complicated expression.

EXAMPLE 10.2 Consider the simple C program shown below.

```
#include <stdio.h>

main()
{
    int u1, u2;
    int v = 3;
    int *pv;           /* pv points to v */
    u1 = 2 * (v + 5);      /* ordinary expression */

    pv = &v;
    u2 = 2 * (*pv + 5);    /* equivalent expression */

    printf("\nu1=%d    u2=%d", u1, u2);
}
```

This program involves the use of two integer expressions. The first, $2 * (v + 5)$, is an ordinary arithmetic expression whereas the second, $2 * (*pv + 5)$, involves the use of a pointer. The expressions are equivalent, since v and $*pv$ each represent the same integer value.

The following output is generated when the program is executed.

```
u1=16    u2=16
```

An indirect reference can also appear on the left side of an assignment statement. This provides another method for assigning a value to a variable or an array element.

EXAMPLE 10.3 A simple C program is shown below.

```
#include <stdio.h>

main()
{
    int v = 3;
    int *pv;

    pv = &v;           /* pv points to v */
    printf("\n*pv=%d    v=%d", *pv, v);

    *pv = 0;          /* reset v indirectly */
    printf("\n\n*pv=%d    v=%d", *pv, v);
}
```

The program begins by assigning an initial value of 3 to the integer variable *v*, and then assigns the address of *v* to the pointer variable *pv*. Thus, *pv* becomes a pointer to *v*. The expression **pv* therefore represents the value 3. The first *printf* statement is intended to illustrate this by displaying the current values of **pv* and *v*.

Following the first *printf* statement, the value of **pv* is reset to 0. Therefore, *v* will be reassigned the value 0. This is illustrated by the second *printf* statement, which causes the new values of **pv* and *v* to be displayed.

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

```
*pv=3    v=3
*pv=0    v=0
```

Thus, the value of *v* has been altered by assigning a new value to **pv*.

Pointer variables can point to numeric or character variables, arrays, functions or other pointer variables. (They can also point to certain other data structures that will be discussed later in this book.) Thus, a pointer variable can be assigned the address of an ordinary variable (e.g., *pv* = *&v*). Also, a pointer variable can be assigned the value of another pointer variable (e.g., *pv* = *px*), provided both pointer variables point to data items of the same type. Moreover, a pointer variable can be assigned a *null* (zero) value, as explained in Sec. 10.2 below. On the other hand, *ordinary* variables *cannot* be assigned arbitrary addresses (i.e., an expression such as *&x* cannot appear on the left-hand side of an assignment statement).

Section 10.5 presents additional information concerning those operations that can be carried out on pointers.

10.2 POINTER DECLARATIONS

Pointer variables, like all other variables, must be declared before they may be used in a C program. The interpretation of a pointer declaration differs, however, from the interpretation of other variable declarations. When a pointer variable is declared, the variable name must be preceded by an asterisk (*). This identifies the fact that the variable is a pointer. The data type that appears in the declaration refers to the *object* of the pointer, i.e., the data item that is stored in the address represented by the pointer, rather than the pointer itself.

Thus, a pointer declaration may be written in general terms as

```
data-type *ptvar;
```

where *ptvar* is the name of the pointer variable, and *data-type* refers to the data type of the pointer's object. Remember that an asterisk must precede *ptvar*.

EXAMPLE 10.4 A C program contains the following declarations.

```
float u, v;
float *pv;
```

The first line declares *u* and *v* to be floating-point variables. The second line declares *pv* to be a pointer variable whose object is a floating-point quantity; i.e., *pv* points to a floating-point quantity. Note that *pv* represents an *address*, not a floating-point quantity. (Some additional pointer declarations are shown in Examples 10.1 to 10.3.)

Within a variable declaration, a pointer variable can be initialized by assigning it the address of another variable. Remember that the variable whose address is assigned to the pointer variable must have been declared earlier in the program.

EXAMPLE 10.5 A C program contains the following declarations.

```
float u, v;
float *pv = &v;
```

The variables *u* and *v* are declared to be floating-point variables and *pv* is declared as a pointer variable that points to a floating-point quantity, as in Example 10.4. In addition, the address of *v* is initially assigned to *pv*.

This terminology can be confusing. Remember that these declarations are equivalent to writing

```
float u, v;           /* floating-point variable declarations */
float *pv;            /* pointer variable declaration */
. . .
pv = &v;             /* assign v's address to pv */
```

Note that an asterisk is not included in the assignment statement.

In general, it does not make sense to assign an integer value to a pointer variable. An exception, however, is an assignment of 0, which is sometimes used to indicate some special condition. In such situations the recommended programming practice is to define a symbolic constant **NULL** which represents 0, and to use **NULL** in the pointer initialization. This practice emphasizes the fact that the zero assignment represents a special situation.

EXAMPLE 10.6 A C program contains the following symbolic constant definitions and array declarations.

```
#define NULL 0
float u, v;
float *pv = NULL;
```

The variables *u* and *v* are declared to be floating-point variables and *pv* is declared as a pointer variable that points to a floating-point quantity. In addition, *pv* is initially assigned a value of 0 to indicate some special condition dictated by the logic of the program (which is not shown in this example). The use of the symbolic constant **NULL** suggests that this initial assignment is something other than the assignment of an ordinary integer value.

We will see other kinds of pointer declarations later in this chapter.

10.3 PASSING POINTERS TO A FUNCTION

Pointers are often passed to a function as arguments. This allows data items within the calling portion of the program to be accessed by the function, altered within the function, and then returned to the calling portion of the program in altered form. We refer to this use of pointers as passing arguments by *reference* (or by *address* or by *location*), in contrast to passing arguments by *value* as discussed in Chap. 7.

When an argument is passed by value, the data item is *copied* to the function. Thus, any alteration made to the data item within the function is not carried over into the calling routine (see Sec. 7.5). When an argument is passed by reference, however (i.e., when a pointer is passed to a function), the *address* of a data item is passed to the function. The contents of that address can be accessed freely, either within the function or within the calling routine. Moreover, any change that is made to the data item (i.e., to the contents of the address) will be recognized in both the function and the calling routine. Thus, the use of a pointer as a function argument permits the corresponding data item to be altered globally from within the function.

When pointers are used as arguments to a function, some care is required with the formal argument declarations within the function. Specifically, formal pointer arguments that must each be preceded by an asterisk. Function prototypes are written in the same manner. If a function declaration does not include variable names, the data type of each pointer argument must be followed by an asterisk. The use of pointer arguments is illustrated in the following example.

EXAMPLE 10.7 Here is a simple C program that illustrates the difference between ordinary arguments, which are passed by value, and pointer arguments, which are passed by reference.

```

#include <stdio.h>

void funct1(int u, int v);           /* function prototype */
void funct2(int *pu, int *pv);       /* function prototype */

main()
{
    int u = 1;
    int v = 3;

    printf("\nBefore calling funct1: u=%d    v=%d", u, v);
    funct1(u, v);
    printf("\nAfter calling funct1:   u=%d    v=%d", u, v);

    printf("\n\nBefore calling funct2:   u=%d    v=%d", u, v);
    funct2(&u, &v);
    printf("\nAfter calling funct2:   u=%d    v=%d", u, v);
}

void funct1(int u, int v)
{
    u = 0;
    v = 0;
    printf("\nWithin funct1:           u=%d    v=%d", u, v);
    return;
}

void funct2(int *pu, int *pv)
{
    *pu = 0;
    *pv = 0;
    printf("\nWithin funct2:           *pu=%d *pv=%d", *pu, *pv);
    return;
}

```

This program contains two functions, called `funct1` and `funct2`. The first function, `funct1`, receives two integer variables as arguments. These variables are originally assigned the values 1 and 3, respectively. The values are then changed, to 0, 0 within `funct1`. The new values are not recognized in `main`, however, because the arguments were passed by value, and any changes to the arguments are local to the function in which the changes occur.

Now consider the second function, `funct2`. This function receives two *pointers* to integer variables as its arguments. The arguments are identified as pointers by the indirection operators (i.e., the asterisks) that appear in the argument declaration. In addition, the argument declaration indicates that the pointers contain the addresses of *integer* quantities.

Within `funct2`, the contents of the pointer addresses are reassigned the values 0, 0. Since the addresses are recognized in both `funct2` and `main`, the reassigned values will be recognized within `main` after the call to `funct2`. Therefore, the integer variables `u` and `v` will have their values changed from 1, 3 to 0, 0.

The six `printf` statements illustrate the values of `u` and `v`, and their associated values `*pu` and `*pv`, within `main` and within the two functions. Hence, the following output is generated when the program is executed.

```

Before calling funct1: u=1    v=3
Within funct1:         u=0    v=0
After calling funct1:  u=1    v=3

```

```

Before calling funct2: u=1 v=3
Within funct2:          *pu=0 *pv=0
After calling funct2:   u=0  v=0

```

Notice that the values of *u* and *v* are unchanged within *main* after the call to *funct1*, though the values of these variables are changed within *main* after the call to *funct2*. Thus, the output illustrates the local nature of the alterations within *funct1*, and the global nature of the alterations within *funct2*.

This example contains some additional features that should be pointed out. Notice, for example, the function prototype

```
void funct2(int *pu, int *pv);
```

The items in parentheses identify the arguments as pointers to integer quantities. The pointer variables, *pu* and *pv*, have not been declared elsewhere in *main*. This is permitted in the function prototype, however, because *pu* and *pv* are dummy arguments rather than actual arguments. The function declaration could also have been written without any variable names, as

```
void funct2(int *, int *);
```

Now consider the declaration of the formal arguments within the first line of *funct2*, i.e.,

```
void funct2(int *pu, int *pv)
```

The formal arguments *pu* and *pv* are consistent with the dummy arguments in the function prototype. In this example the corresponding variable names are the same, though this is generally not required.

Finally, notice the manner in which *u* and *v* are accessed within *funct2*, i.e.,

```
*pu = 0;
*pv = 0;
```

Thus, *u* and *v* are accessed indirectly, by referencing the contents of the addresses represented by the pointers *pu* and *pv*. This is necessary because the variables *u* and *v* are not recognized as such within *funct2*.

We have already mentioned the fact that an array name is actually a pointer to the array; i.e., the array name represents the address of the first element in the array (see Sec. 9.3). Therefore, an array name is treated as a pointer when it is passed to a function. However, it is not necessary to precede the array name with an ampersand within the function call.

An array name that appears as a formal argument within a function definition can be declared either as a pointer or as an array of unspecified size, as shown in Sec. 9.3. The choice is a matter of personal preference, though it will often be determined by the manner in which the individual array elements are accessed within the function (more about this in the next section).

EXAMPLE 10.8 Analyzing a Line of Text Suppose we wish to analyze a line of text by examining each of the characters and determining into which of several different categories it falls. In particular, suppose we count the number of vowels, consonants, digits, whitespace characters and “other” characters (punctuation, operators, brackets, etc.) This can easily be accomplished by reading in a line of text, storing it in a one-dimensional character array, and then analyzing the individual array elements. An appropriate counter will be incremented for each character. The value of each counter (number of vowels, number of consonants, etc.) can then be written out after all of the characters have been analyzed.

Let us write a complete C program that will carry out such an analysis. To do so, we first define the following symbols.

```

line = an 80-element character array containing the line of text
vowels = an integer counter indicating the number of vowels
consonants = an integer counter indicating the number of consonants
digits = an integer counter indicating the number of digits

```

whitespc = an integer counter indicating the number of whitespace characters (blank spaces or tabs)

other = an integer counter indicating the number of characters that do not fall into any of the preceding categories

Notice that newline characters are not included in the "whitespace" category, because there can be no newline characters within a single line of text.

We will structure the program so that the line of text is read into the main portion of the program, and then passed to a function where it will be analyzed. The function will return the value of each counter after all of the characters have been analyzed. The results of the analysis (i.e., the value of each counter) will then be displayed from the main portion of the program.

The actual analysis can be carried out by creating a loop to examine each of the characters. Within the loop we first convert each character that is a letter to uppercase. This avoids the need to distinguish between uppercase and lowercase letters. We can then categorize the character using a nest of if - else statements. Once the proper category has been identified, the corresponding counter is incremented. The entire process is repeated until the string termination character (\0) has been found.

The complete C program is shown below.

```

/* count the number of vowels, consonants, digits, whitespace characters,
   and "other" characters in a line of text */
```

```

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

/* function prototype */
void scan_line(char line[], int *pv, int *pc, int *pd, int *pw, int *po);

main()
{
    char line[80];           /* line of text */
    int vowels = 0;          /* vowel counter */
    int consonants = 0;      /* consonant counter */
    int digits = 0;          /* digit counter */
    int whitespc = 0;        /* whitespace counter */
    int other = 0;           /* remaining character counter */

    printf("Enter a line of text below:\n");
    scanf("%[^\\n]", line);

    scan_line(line, &vowels, &consonants, &digits, &whitespc, &other);

    printf("\nNo. of vowels: %d", vowels);
    printf("\nNo. of consonants: %d", consonants);
    printf("\nNo. of digits: %d", digits);
    printf("\nNo. of whitespace characters: %d", whitespc);
    printf("\nNo. of other characters: %d", other);
}

void scan_line(char line[], int *pv, int *pc, int *pd, int *pw, int *po)
/* analyze the characters in a line of text */

{
    char c;                 /* uppercase character */
    int count = 0;           /* character counter */

```

```

while ((c = toupper(line[count])) != '\0') {
    if (c == 'A' || c == 'E' || c == 'I' || c == 'O' || c == 'U')
        ++ *pv;                                /* vowel */
    else if (c >= 'A' && c <= 'Z')
        ++ *pc;                                /* consonant */
    else if (c >= '0' && c <= '9')
        ++ *pd;                                /* digit */
    else if (c == ' ' || c == '\t')
        ++ *pw;                                /* whitespace */
    else
        ++ *po;                                /* other */

    ++count;
}
return;
}

```

Notice the function prototype for `scan_line` that appears at the beginning of the program. In particular, notice the use of the `void` data type, and notice the manner in which the argument data types are specified. Note the distinction between the array argument and the remaining pointer arguments.

Also, observe the manner in which the actual arguments are written in the call to `scan_line`. The array argument, `line`, is not preceded by an ampersand, since arrays are, by definition, pointers. Each of the remaining arguments must be preceded by an ampersand so that its address, rather than its value, is passed to the function.

Now consider the function `scan_line`. All of the formal arguments, including `line`, are pointers. However, `line` is declared as an array whose size is unspecified, whereas the remaining arguments are specifically declared as pointers. It is possible (and quite common) to declare `line` as a pointer rather than an array. Thus, the first line of `scan_line` could have been written as

```
void scan_line(char *line, int *pv, int *pc, int *pd, int *pw, int *po)
```

rather than as shown in the program listing. To be consistent, the corresponding function prototype would then be written in a similar manner.

Incrementing the various counters also requires some explanation. First, note that it is the *content* of each address (i.e., the *object* of each pointer) that is incremented. Second, note that each indirection expression (e.g., `*pv`) is *preceded* by the unary operator `++`. Since the unary operators are evaluated from right to left, we are assured that the content of each address, rather than the address itself, is increased in value.

Here is a typical dialog that might be encountered when the program is executed. (The line of text entered by the user is underlined.)

Enter a line of text below:

Personal computers with memories in excess of 4096 KB are now quite common.

The corresponding output is:

```
No. of vowels: 23
No. of consonants: 35
No. of digits: 4
No. of whitespace characters: 12
No. of other characters: 1
```

Thus, we see that this particular line of text contains 23 vowels, 35 consonants, 4 digits, 12 whitespace characters (blank spaces), and one other character (the period).

Recall that the `scanf` function requires those arguments that are ordinary variables to be preceded by ampersands (see Sec. 4.4). However, array names are exempted from this requirement. This may have seemed somewhat mysterious back in Chap. 4 but it should now make sense, considering what we now know about array names and addresses. Thus, the `scanf` function requires that the *addresses* of the data items being entered into the computer's memory be specified. The ampersands provide a means for accessing the addresses of ordinary single-valued variables. Ampersands are not required with array names, since array names themselves represent addresses.

EXAMPLE 10.9 The skeletal structure of a C program is shown below (repeated from Example 4.5).

```
#include <stdio.h>

main()
{
    char item[20];
    int partno;
    float cost;

    . . .

    scanf("%s %d %f", item, &partno, &cost);

    . . .
}
```

The `scanf` statement causes a character string, an integer quantity and a floating-point quantity to be entered into the computer and stored in the addresses associated with `item`, `partno` and `cost`, respectively. Since `item` is the name of an array, it is understood to represent an address. Hence, `item` need not (cannot) be preceded by an ampersand within the `scanf` statement. On the other hand, `partno` and `cost` are conventional variables. Therefore they must be written as `&partno` and `&cost` within the `scanf` statement. The ampersands are required in order to access the *addresses* of these variables rather than their values.

If the `scanf` function is used to enter a single array element rather than an entire array, the name of the array element must be preceded by an ampersand, as shown below (from Example 9.8):

```
scanf("%f", &list[count]);
```

It is possible to pass a *portion* of an array, rather than an entire array, to a function. To do so, the address of the first array element to be passed must be specified as an argument. The remainder of the array, starting with the specified array element, will then be passed to the function.

EXAMPLE 10.10 The skeletal structure of a C program is shown below.

```
#include <stdio.h>

void process(float z[]);

main()
{
    float z[100];
    . . .
    /* enter values for elements of z */
    . . .
    process(&z[50]);
    . . .
}
```

```

void process(float f[])
{
    . . .

    /* process elements of f */

    . . .

    return;
}

```

Within `main`, `z` is declared to be a 100-element, floating-point array. After the elements of `z` are entered into the computer, the address of `z[50]` (i.e., `&z[50]`) is passed to the function `process`. Hence, the last 50 elements of `z` (i.e., the elements `z[50]` through `z[99]`) will be available to `process`.

In the next section we will see that the address of `z[50]` can be written as `z + 50` rather than `&z[50]`. Therefore, the call to `process` can appear as `process(z + 50)` rather than `process(&z[50])`, as shown above. Either method may be used, depending on the programmer's preferences.

Within `process`, the corresponding array is referred to as `f`. This array is declared to be a floating-point array whose size is unspecified. Thus, the fact that the function receives only a portion of `z` is immaterial; if all of the array elements are altered within `process`, only the last 50 elements will be affected within `main`.

Within `process`, it may be desirable to declare the formal argument `f` as a pointer to a floating-point quantity rather than an array name. Thus, the outline of `process` may be written as

```

void process(float *f)
{
    . . .

    /* process elements of f */

    . . .

    return;
}

```

Notice the difference between the formal argument declarations in the two function outlines. Both declarations are valid.

A function can also return a pointer to the calling portion of the program. To do so, the function definition and any corresponding function declarations must indicate that the function will return a pointer. This is accomplished by preceding the function name by an asterisk. The asterisk must appear in both the function definition and the function declarations.

EXAMPLE 10.11 Shown below is the skeletal structure of a C program that transfers a double-precision array to a function and returns a pointer to one of the array elements.

```

#include <stdio.h>

double *scan(double z[]);

main()
{
    double z[100];           /* array declaration */
    double *pz;              /* pointer declaration */

    /* enter values for elements of z */

    . . .

    pz = scan(z);

    . . .
}

```

```

    double *scan(double f[])
{
    double *pf;           /* pointer declaration */

    . . .

    /* process elements of f */

    pf = . . . . .;
    return(pf);
}

```

Within `main` we see that `z` is declared to be a 100-element, double-precision array, and `pz` is a pointer to a double-precision quantity. We also see a declaration for the function `scan`. Note that `scan` will accept a double-precision array as an argument, and it will return a pointer to (i.e., the address of) a double-precision quantity. The asterisk preceding the function name (`*scan`) indicates that the function will return a pointer.

Within the function definition, the first line indicates that `scan` accepts one formal parameter (`f[]`) and returns a pointer to a double-precision quantity. The formal parameter will be a one-dimensional, double-precision array. The outline suggests that the address of one of the array elements is assigned to the pointer `pf` during or after the processing of the array elements. This address is then returned to `main`, where it is assigned to the pointer variable `pz`.

10.4 POINTERS AND ONE-DIMENSIONAL ARRAYS

Recall that an array name is really a pointer to the first element in the array. Therefore, if `x` is a one-dimensional array, then the address of the first array element can be expressed as either `&x[0]` or simply as `x`. Moreover, the address of the second array element can be written as either `&x[1]` or as `(x + 1)`, and so on. In general, the address of array element $(i + 1)$ can be expressed as either `&x[i]` or as `(x + i)`. Thus we have two different ways to write the address of any array element: We can write the actual array element, preceded by an ampersand; or we can write an expression in which the subscript is added to the array name.

In the latter case, it should be understood that we are dealing with a very special and unusual type of expression. In the expression `(x + i)`, for example, `x` represents an address, whereas `i` represents an integer quantity. Moreover, `x` is the name of an array whose elements may be characters, integers, floating-point quantities, etc. (though all of the the array elements must be of the same data type). Thus, we are not simply adding numerical values. Rather, we are specifying an address that is a certain number of memory cells beyond the address of the first array element. Or, in simpler terms, we are specifying a location that is `i` array elements beyond the first. Hence, the expression `(x + i)` is a symbolic representation for an address specification rather than an arithmetic expression.

Recall that the number of memory cells associated with an array element will depend upon the data type of the array as well as the particular computer's architecture. With some computers, for example, an integer quantity occupies two bytes (two memory cells), a floating-point quantity requires four bytes, and a double-precision quantity requires eight bytes of memory. With other computers, an integer quantity may require four bytes, and floating-point and double-precision quantities may each require eight bytes. And so on.

When writing the address of an array element in the form `(x + i)`, however, you need not be concerned with the number of memory cells associated with each type of array element; the C compiler adjusts for this automatically. You must specify only the address of the first array element (i.e., the name of the array) and the number of array elements beyond the first (i.e., a value for the subscript). The value of `i` is sometimes referred to as an *offset* when used in this manner.

Since `&x[i]` and `(x + i)` both represent the address of the i^{th} element of `x`, it would seem reasonable that `x[i]` and `*(x + i)` both represent the contents of that address, i.e., the *value* of the i^{th} element of `x`. This is indeed the case. The two terms are interchangeable. Hence, either term can be used in any particular application. The choice depends upon your individual preferences.

EXAMPLE 10.12 Here is a simple program that illustrates the relationship between array elements and their addresses.

```
#include <stdio.h>

main()
{
    static int x[10] = {10, 11, 12, 13, 14, 15, 16, 17, 18, 19};
    int i;

    for (i = 0; i <= 9; ++i) {
        /* display an array element */
        printf("\ni= %d      x[i]= %d      *(x+i)= %d", i, x[i], *(x+i));
        /* display the corresponding array address */
        printf("      &x[i]= %X      x+i= %X", &x[i], (x+i));
    }
}
```

This program defines a one-dimensional, 10-element integer array *x*, whose elements are assigned the values 10, 11, . . . , 19. The action portion of the program consists of a loop that displays the value and the corresponding address of each array element. Note that the value of each array element is specified in two different ways, as *x[i]* and as **(x+i)*, in order to illustrate their equivalence. Similarly, the address of each array element is specified in two different ways, as *&x[i]* and as *(x+i)*, for the same reason. Therefore the value and the address of each array element should appear twice.

Execution of this program results in the following output.

| | | | | |
|------|----------|------------|-----------|---------|
| i= 0 | x[i]= 10 | *(x+i)= 10 | &x[i]= 72 | x+i= 72 |
| i= 1 | x[i]= 11 | *(x+i)= 11 | &x[i]= 74 | x+i= 74 |
| i= 2 | x[i]= 12 | *(x+i)= 12 | &x[i]= 76 | x+i= 76 |
| i= 3 | x[i]= 13 | *(x+i)= 13 | &x[i]= 78 | x+i= 78 |
| i= 4 | x[i]= 14 | *(x+i)= 14 | &x[i]= 7A | x+i= 7A |
| i= 5 | x[i]= 15 | *(x+i)= 15 | &x[i]= 7C | x+i= 7C |
| i= 6 | x[i]= 16 | *(x+i)= 16 | &x[i]= 7E | x+i= 7E |
| i= 7 | x[i]= 17 | *(x+i)= 17 | &x[i]= 80 | x+i= 80 |
| i= 8 | x[i]= 18 | *(x+i)= 18 | &x[i]= 82 | x+i= 82 |
| i= 9 | x[i]= 19 | *(x+i)= 19 | &x[i]= 84 | x+i= 84 |

The output clearly illustrates the distinction between *x[i]*, which represents the value of the *i*th array element, and *&x[i]*, which represents its address. Moreover, we see that the value of the *i*th array element can be represented by either *x[i]* or **(x+i)*, and the address of the *i*th element can be represented by either *&x[i]* or *x+i*. Thus we see another comparison, between **(x+i)*, which also represents the value of the *i*th element, and *x+i*, which also represents its address.

Notice, for example, that the first array element (corresponding to *i* = 0) has been assigned a value of 10 and a (hexadecimal) address of 72. The second array element has a value of 11 and an address of 74, etc. Thus, memory location 72 will contain the integer value 10, location 74 will contain 11, and so on.

You should understand that these addresses are assigned automatically by the compiler.

When assigning a *value* to an array element such as *x[i]*, the left side of the assignment statement may be written as either *x[i]* or as **(x + i)*. Thus, a value may be assigned directly to an array element, or it may be assigned to the memory area whose address is that of the array element.

On the other hand, it is sometimes necessary to assign an *address* to an identifier. In such situations, a pointer variable must appear on the left side of the assignment statement. It is *not* possible to assign an arbitrary address to an array name or to an array element. Thus, expressions such as *x*, *(x + i)* and *&x[i]* cannot appear on the left side of an assignment statement. Moreover, the address of an array cannot arbitrarily be altered, so that expressions such as *++x* are not permitted.

EXAMPLE 10.13 Consider the skeletal structure of the C program shown below.

```
#include <stdio.h>

main()
{
    int line[80];
    int *pl;

    . . .

    /* assign values */
    line[2] = line[1];
    line[2] = *(line + 1);
    *(line + 2) = line[1];
    *(line + 2) = *(line + 1);

    /* assign addresses */
    pl = &line[1];
    pl = line + 1;
}
```

Each of the first four assignment statements assigns the value of the second array element (i.e., `line[1]`) to the third array element (`line[2]`). Thus, the four statements are all equivalent. An experienced programmer would probably choose either the first or the fourth, however, in order that the notation be consistent.

The last two assignment statements each assigns the *address* of the second array element to the pointer `pl`. We might choose to do this in an actual program if it were necessary to “tag” the address of `line[1]` for some reason.

Note that *the address of one array element cannot be assigned to some other array element*. Thus we *cannot* write a statement such as

```
&line[2] = &line[1];
```

On the other hand, we *can* assign the *value* of one array element to another through a pointer if we wish, e.g.,

```
pl = &line[1];
line[2] = *pl;
```

or

```
pl = line + 1;
*(line + 2) = *pl;
```

If a numerical array is defined as a pointer variable, the array elements cannot be assigned initial values. Therefore, a conventional array definition is required if initial values will be assigned to the elements of a numerical array. However, a *character-type* pointer variable can be assigned an entire string as a part of the variable declaration. Thus, a string can conveniently be represented by either a one-dimensional character array or a character pointer.

EXAMPLE 10.14 Shown below is a simple C program in which two strings are represented as one-dimensional character arrays.

```
#include <stdio.h>

char x[] = "This string is declared externally\n\n";
```

```

main()
{
    static char y[] = "This string is declared within main";
    printf("%s", x);
    printf("%s", y);
}

```

The first string is assigned to the external array `x[]`. The second string is assigned to the static array `y[]`, which is defined within `main`. This second definition occurs within a function; therefore `y[]` must be defined as a `static` array so that it can be initialized.

Here is a different version of the same program. The strings are now assigned to pointer variables rather than to one-dimensional arrays.

```

#include <stdio.h>

char *x = "This string is declared externally\n\n";

main()
{
    char *y = "This string is declared within main";

    printf("%s", x);
    printf("%s", y);
}

```

The external pointer variable `x` points to the beginning of the first string, whereas the pointer variable `y`, declared within `main`, points to the beginning of the second string. Note that `y` can now be initialized without being declared `static`.

Execution of either program produces the following output.

```

This string is declared externally

This string is declared within main

```

Syntactically, of course, it is possible to declare a pointer variable `static`. However, there is no reason to do so in this example.

10.5 DYNAMIC MEMORY ALLOCATION

Since an array name is actually a pointer to the first element within the array, it should be possible to define the array as a pointer variable rather than as a conventional array. Syntactically, the two definitions are equivalent. However, a conventional array definition results in a fixed block of memory being reserved at the beginning of program execution, whereas this does not occur if the array is represented in terms of a pointer variable. Therefore, the use of a pointer variable to represent an array requires some type of initial memory assignment before the array elements are processed. This is known as *dynamic memory allocation*. Generally, the `malloc` library function is used for this purpose, as illustrated in the next example.

EXAMPLE 10.15 Suppose `x` is a one-dimensional, 10-element array of integers. It is possible to define `x` as a pointer variable rather than an array. Thus, we can write

```
int *x;
```

rather than

```
int x[10];
```

or

```
#define SIZE 10
int x[SIZE];
```

However, *x* is not automatically assigned a memory block when it is defined as a pointer variable, though a block of memory large enough to store 10 integer quantities will be reserved in advance when *x* is defined as an array.

To assign sufficient memory for *x*, we can make use of the library function **malloc**, as follows.

```
x = (int *) malloc(10 * sizeof(int));
```

This function reserves a block of memory whose size (in bytes) is equivalent to 10 integer quantities. As written, the function returns a pointer to an integer. This pointer indicates the beginning of the memory block. In general, the type cast preceding **malloc** must be consistent with the data type of the pointer variable. Thus, if *y* were defined as a pointer to a double-precision quantity and we wanted enough memory to store 10 double-precision quantities, we would write

```
y = (double *) malloc(10 * sizeof(double));
```

If the declaration is to include the assignment of initial values, however, then x must be defined as an array rather than a pointer variable. For example,

```
int x[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

or

```
int x[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

When programming in C, it is not unusual to use pointer expressions rather than references to individual array elements. The resulting programs may appear strange at first, though they are straightforward once you become comfortable accessing values that are stored in specified addresses. Generally, a small amount of practice is all that is required.

EXAMPLE 10.16 Reordering a List of Numbers To illustrate the use of pointers with dynamic memory allocation, let us once again consider the problem of reordering a list of integers, as described in Example 9.13. Now, however, we will utilize pointer expressions to access numerical values rather than refer explicitly to individual array elements. In all other respects, we present a program that is identical to that given in Example 9.13.

Here is the complete C program.

```
/* reorder a one-dimensional, integer array from smallest to largest,
   using pointer notation */
```

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

void reorder(int n, int *x);

main()
{
    int i, n, *x;

    /* read in a value for n */
    printf("\nHow many numbers will be entered? ");
    scanf("%d", &n);
    printf("\n");

    /* allocate memory */
    x = (int *) malloc(n * sizeof(int));
```

```

/* read in the list of numbers */
for (i = 0; i < n; ++i) {
    printf("i = %d  x = ", i + 1);
    scanf("%d", x + i);
}

/* reorder all array elements */
reorder(n, x);

/* display the reordered list of numbers */
printf("\n\nReordered List of Numbers:\n\n");
for (i = 0; i < n; ++i)
    printf("i = %d  x = %d\n", i + 1, *(x + i));
}

void reorder(int n, int *x) /* rearrange the list of numbers */
{
    int i, item, temp;
    for (item = 0; item < n - 1; ++item)

        /* find the smallest of all remaining elements */
        for (i = item + 1; i < n; ++i)

            if (*(x + i) < *(x + item)) {

                /* interchange two elements */
                temp = *(x + item);
                *(x + item) = *(x + i);
                *(x + i) = temp;
            }
    return;
}

```

In this program, the integer array is defined as a pointer to an integer. Memory is initially assigned to the pointer variable via the `malloc` library function. Elsewhere in the program, pointer notation is used to process the individual array elements. For example, the function prototype now specifies that the second argument is a pointer to an integer quantity rather than an integer array. This pointer will identify the beginning of the integer array.

We also see that the `scanf` function now specifies the address of the *i*th element as `x + i` rather than `&x[i]`. Similarly, the `printf` function now represents the value of the *i*th element as `*(x + i)` rather than `x[i]`. The call to `reorder`, however, is the same as in the earlier program—namely, `reorder(n, x)`.

Within the function `reorder`, we see that the second formal argument is now defined as a pointer variable rather than an integer array. This is consistent with the function prototype. Even more pronounced, however, are the differences in the `if` statement. In particular, notice that each reference to an array element is now written as the contents of an address. Thus `x[i]` is now written as `*(x + i)`, and `x[item]` is now written as `*(x + item)`. This compound `if` statement can be viewed as a conditional interchange involving the contents of two different addresses, rather than an interchange of two different elements within a conventional array.

You should compare this program with that shown in Example 9.13 in order to appreciate the differences. Both programs will generate identical results with the same input data. However, you should understand the syntactic differences between the two programs.

An important advantage of dynamic memory allocation is the ability to reserve as much memory as may be required during program execution, and then release this memory when it is no longer needed. Moreover, this process may be repeated many times during execution of a program. The library functions `malloc` and `free` are used for these purposes, as illustrated in Example 11.32 (see Sec. 11.6).

10.6 OPERATIONS ON POINTERS

In Sec. 10.4 we saw that an integer value can be added to an array name in order to access an individual array element. The integer value is interpreted as an array subscript; it represents the location of the desired array element relative to the first element in the array. This works because all of the array elements are of the same data type, and therefore each array element occupies the same number of memory cells (i.e., the same number of bytes or words). The actual number of memory cells that separate the two array elements will depend on the data type of the array, though this is taken care of automatically by the compiler and therefore need not concern you directly.

This concept can be extended to pointer variables in general. Thus, an integer value can be added to or subtracted from a pointer variable, though the resulting expression must be interpreted very carefully. Suppose, for example, that px is a pointer variable that represents the address of some variable x . We can write expressions such as $++\text{px}$, $--\text{px}$, $(\text{px} + 3)$, $(\text{px} + i)$, and $(\text{px} - i)$, where i is an integer variable. Each expression will represent an address that is located some distance from the original address represented by px . The exact distance will be the product of the integer quantity and the number of bytes or words associated with the data item to which px points. Suppose, for example, that px points to an integer quantity, and each integer quantity requires two bytes of memory. Then the expression $(\text{px} + 3)$ will result in an address that is six bytes beyond the integer to which px points, as illustrated in Fig. 10.3. It should be understood, however, that this new address will *not* necessarily represent the address of another data item, particularly if the data items stored between the two addresses involve different data types.

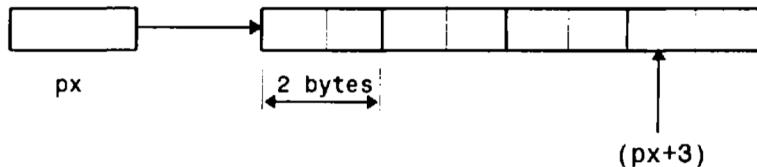


Fig. 10.3

EXAMPLE 10.17 Consider the simple C program shown below.

```
#include <stdio.h>

main()
{
    int *px;          /* pointer to an integer */
    int i = 1;
    float f = 0.3;
    double d = 0.005;
    char c = '*';

    px = &i;
    printf("Values: i=%i f=%f d=%f c=%c\n\n", i, f, d, c);
    printf("Addresses: &i=%X &f=%X &d=%X &c=%X\n\n", &i, &f, &d, &c);
    printf("Pointer values: px=%X px + 1=%X px + 2=%X px + 3=%X",
           px, px + 1, px + 2, px + 3);
}
```

This program displays the values and addresses associated with four different types of variables: i , an integer variable; f , a floating-point variable; d , a double-precision variable; and c , a character variable. The program also makes use of a pointer variable, px , which represents the address of i . The values of px , $\text{px} + 1$, $\text{px} + 2$ and $\text{px} + 3$ are also displayed, so that they may be compared with the addresses of the different variables.

Execution of the program results in the following output.

```
Values:   i=1    f=0.300000  d=0.005000  c=*
Addresses:  &i=117E  &f=1180  &d=1186  &c=118E
Pointer values:  px=117E  px + 1=1180  px + 2=1182  px + 3=1184
```

The first line simply displays the values of the variables, and the second line displays their addresses, as assigned by the compiler. Notice that the number of bytes associated with each data item is different. Thus, the integer value represented by *i* requires two bytes (specifically, addresses 117E and 117F). The floating-point value represented by *f* appears to be assigned six bytes (addresses 1180 through 1185), though only four bytes (addresses 1180 through 1183) are actually used for this purpose. (Compilers allocate memory space according to their own rules.) However, eight bytes are required for the double-precision value represented by *d* (addresses 1186 through 118D). And finally, the character represented by *c* begins in address 118E. Only one byte is required to store this single character, though the output does not indicate the number of bytes between this character and the next data item.

Now consider the third line of output, which contains the addresses represented by the pointer expressions. Clearly, *px* represents the address of *i* (i.e., 117E). This comes as no surprise, since this address was explicitly assigned to *px* by the expression *px = &i*. However, *px + 1* moves over only two bytes, to 1180, and *px + 2* moves over another two bytes, to 1182, and so on. The reason is that *px* points to an integer quantity, and integer quantities each require two bytes with this particular C compiler. Therefore, when integer constants are added to *px*, the constants are interpreted in terms of two-byte multiples.

If *px* is defined as a pointer to a different type of object (e.g., a character or a floating-point quantity), then any integer constant that is added to or subtracted from the pointer will be interpreted differently. In particular, each integer value will represent an equivalent number of individual bytes if *px* points to a character, or a corresponding number of four-byte multiples if *px* points to a floating-point quantity. You are encouraged to verify this on your own.

One pointer variable can be subtracted from another provided both variables point to elements of the same array. The resulting value indicates the number of words or bytes separating the corresponding array elements.

EXAMPLE 10.18 In the program shown below, two different pointer variables point to the first and the last elements of an integer array.

```
#include <stdio.h>

main()
{
    int *px, *py;          /* integer pointers */
    static int a[6] = {1, 2, 3, 4, 5, 6};

    px = &a[0];
    py = &a[5];
    printf("px=%X  py=%X", px, py);
    printf("\n\npy - px=%X", py - px);
}
```

In particular, the pointer variable *px* points to *a[0]*, and *py* points to *a[5]*. The difference, *py - px*, should be 5, since *a[5]* is the fifth element beyond *a[0]*.

Execution of the program results in the following output.

```
px=52  py=5C
py - px=5
```

The first line indicates that the address of *a[0]* is 52, and the address of *a[5]* is 5C. The difference between these two hexadecimal numbers is 10 (when converted to decimal). Thus, *a[5]* is stored at an address which is 10 bytes beyond the

address of `a[0]`. Since each integer quantity occupies two bytes, we would expect the difference between `py` and `px` to be $10/2 = 5$. The second line of output confirms this value.

Pointer variables can be compared provided both variables are of the same data type. Such comparisons can be useful when both pointer variables point to elements of the same array. The comparisons can test for either equality or inequality. Moreover, a pointer variable can be compared with zero (which is usually expressed as `NULL` when used in this manner, as explained in Sec. 10.2).

EXAMPLE 10.19 Suppose `px` and `py` are pointer variables that point to elements of the same array. Several logical expressions involving these two variables are shown below. All of the expressions are syntactically correct.

```
(px < py)
(px >= py)
(px == py)
(px != py)
(px == NULL)
```

These expressions can be used as any other logical expression. For example,

```
if (px < py)
    printf("px < py");
else
    printf("px >= py");
```

Expressions such as `(px < py)` indicate whether or not the element associated with `px` is ranked ahead of the element associated with `py` (i.e., whether or not the subscript associated with `*px` is less than the subscript associated with `*py`).

You should understand that the operations discussed previously are the *only* operations that can be carried out on pointers. These permissible operations are summarized below.

1. A pointer variable can be assigned the address of an ordinary variable (e.g., `pv = &v`).
2. A pointer variable can be assigned the value of another pointer variable (e.g., `pv = px`) provided both pointers point to objects of the same data type.
3. A pointer variable can be assigned a null (zero) value (e.g., `pv = NULL`, where `NULL` is a symbolic constant that represents the value 0).
4. An integer quantity can be added to or subtracted from a pointer variable (e.g., `pv + 3`, `++pv`, etc.)
5. One pointer variable can be subtracted from another provided both pointers point to elements of the same array.
6. Two pointer variables can be compared provided both pointers point to objects of the same data type.

Other arithmetic operations on pointers are not allowed. Thus, a pointer variable cannot be multiplied by a constant; two pointer variables cannot be added; and so on. Also, you are again reminded that an ordinary variable cannot be assigned an arbitrary address (i.e., an expression such as `&x` cannot appear on the left side of an assignment statement).

10.7 POINTERS AND MULTIDIMENSIONAL ARRAYS

Since a one-dimensional array can be represented in terms of a pointer (the array name) and an offset (the subscript), it is reasonable to expect that a multidimensional array can also be represented with an equivalent pointer notation. This is indeed the case. A two-dimensional array, for example, is actually a collection of

one-dimensional arrays. Therefore, we can define a two-dimensional array as a pointer to a group of contiguous one-dimensional arrays. Thus, a two-dimensional array declaration can be written as

```
data-type (*ptvar)[expression 2];
```

rather than

```
data-type array[expression 1][expression 2];
```

This concept can be generalized to higher-dimensional arrays; that is,

```
data-type (*ptvar)[expression 2][expression 3] . . . [expression n];
```

replaces

```
data-type array[expression 1][expression 2] . . . [expression n];
```

In these declarations *data-type* refers to the data type of the array, *ptvar* is the name of the pointer variable, *array* is the corresponding array name, and *expression 1*, *expression 2*, . . . , *expression n* are positive-valued integer expressions that indicate the maximum number of array elements associated with each subscript.

Notice the parentheses that surround the array name and the preceding asterisk in the pointer version of each declaration. These parentheses must be present. Without them we would be defining an array of pointers rather than a pointer to a group of arrays, since these particular symbols (i.e., the square brackets and the asterisk) would normally be evaluated right to left. We will say more about this in the next section.

EXAMPLE 10.20 Suppose *x* is a two-dimensional integer array having 10 rows and 20 columns. We can declare *x* as

```
int (*x)[20];
```

rather than

```
int x[10][20];
```

In the first declaration, *x* is defined to be a pointer to a group of contiguous, one-dimensional, 20-element integer arrays. Thus, *x* points to the first 20-element array, which is actually the first row (i.e., row 0) of the original two-dimensional array. Similarly, (*x* + 1) points to the second 20-element array, which is the second row (row 1) of the original two-dimensional array, and so on, as illustrated in Fig. 10.4.

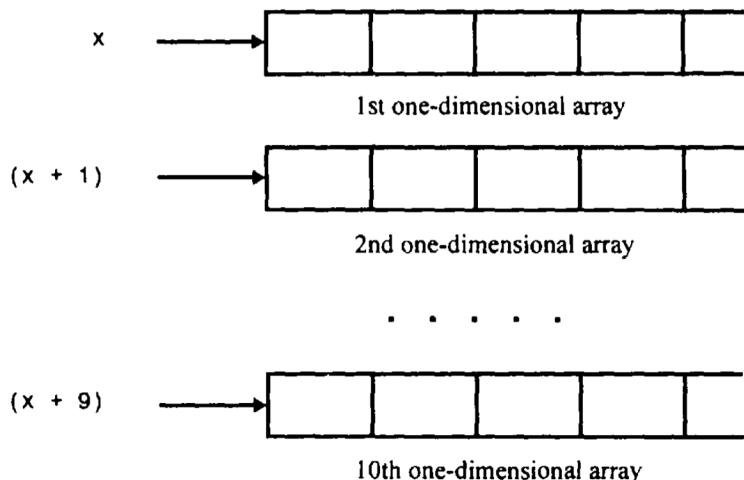


Fig. 10.4

Now consider a three-dimensional floating-point array t . This array can be defined as

```
float (*t)[20][30];
```

rather than

```
float t[10][20][30];
```

In the first declaration, t is defined as a pointer to a group of contiguous two-dimensional, 20×30 floating-point arrays. Hence, t points to the first 20×30 array, $(t + 1)$ points to the second 20×30 array, and so on.

An individual array element within a multidimensional array can be accessed by the repeated use of the indirection operator. Usually, however, this procedure is more awkward than the conventional method for accessing an array element. The following example illustrates the use of the indirection operator.

EXAMPLE 10.21 Suppose x is a two-dimensional integer array having 10 rows and 20 columns, as declared in the previous example. The item in row 2, column 5 can be accessed by writing either

```
x[2][5]
```

or

```
*(*(x + 2) + 5)
```

The second form requires some explanation. First, note that $(x + 2)$ is a pointer to row 2. Therefore the object of this pointer, $*(x + 2)$, refers to the entire row. Since row 2 is a one-dimensional array, $*(x + 2)$ is actually a pointer to the first element in row 2. We now add 5 to this pointer. Hence, $(*(x + 2) + 5)$ is a pointer to element 5 (i.e., the sixth element) in row 2. The object of this pointer, $*(*((x + 2) + 5))$, therefore refers to the item in column 5 of row 2, which is $x[2][5]$. These relationships are illustrated in Fig. 10.5.

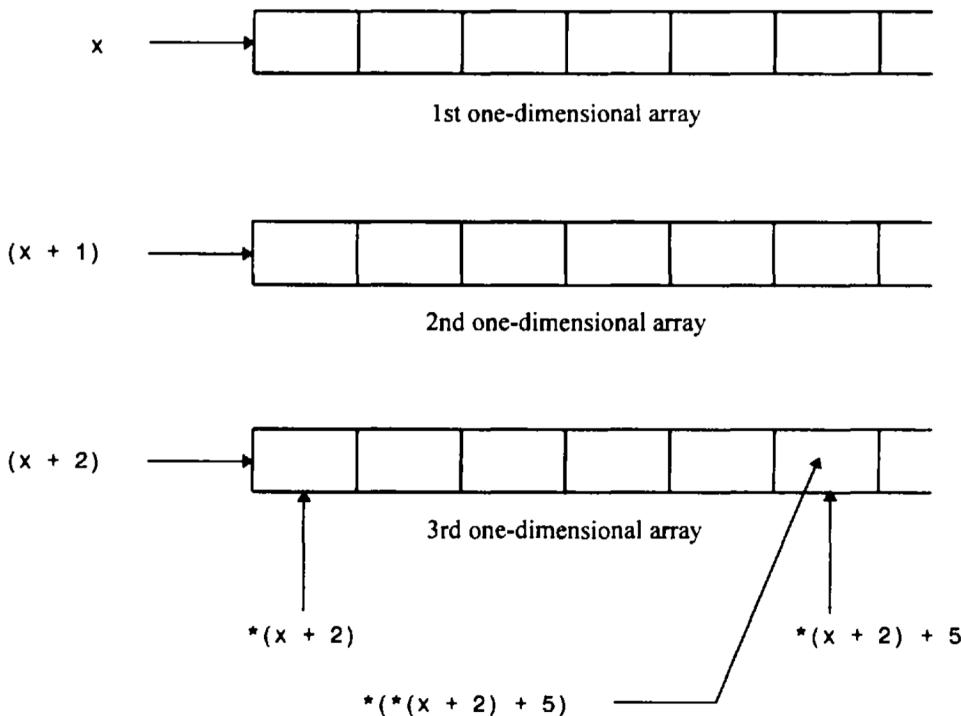


Fig. 10.5

Programs that make use of multidimensional arrays can be written in several different ways. In particular, there are different ways to define the arrays, and different ways to process the individual array elements. The choice of one method over another is often a matter of personal preference. In applications involving numerical arrays, it is often easier to define the arrays in the conventional manner, thus avoiding any possible subtleties associated with initial memory assignments. The following example, however, illustrates the use of pointer notation to process multidimensional numerical arrays.

EXAMPLE 10.22 Adding Two Tables of Numbers In Example 9.19 we developed a C program to calculate the sum of the corresponding elements of two tables of integers. That program required three separate two-dimensional arrays, which were defined and processed in the conventional manner. Here is a variation of the program, in which each two-dimensional array is defined as an array of pointers to a set of one-dimensional integer arrays.

```

/* calculate the sum of the elements in two tables of integers */

/* each 2-dimensional array is processed as an array
   of pointers to a set of 1-dimensional integer arrays */

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

#define MAXROWS 20

/* function prototypes */
void readinput (int *a[MAXROWS], int nrows, int ncols);
void computesums(int *a[MAXROWS], int *b[MAXROWS],
                 int *c[MAXROWS], int nrows, int ncols);
void writeoutput(int *c[MAXROWS], int nrows, int ncols);

main()
{
    int row, nrows, ncols;

    /* pointer definitions */
    int *a[MAXROWS], *b[MAXROWS], *c[MAXROWS];

    printf("How many rows? ");
    scanf("%d", &nrows);
    printf("How many columns? ");
    scanf("%d", &ncols);

    /* allocate initial memory */
    for (row = 0; row < nrows; ++row) {
        a[row] = (int *) malloc (ncols * sizeof(int));
        b[row] = (int *) malloc (ncols * sizeof(int));
        c[row] = (int *) malloc (ncols * sizeof(int));
    }

    printf("\n\nFirst table:\n");
    readinput(a, nrows, ncols);

    printf("\n\nSecond table:\n");
    readinput(b, nrows, ncols);

    computesums(a, b, c, nrows, ncols);

    printf("\n\nSums of the elements:\n\n");
    writeoutput(c, nrows, ncols);
}

```

```

void readininput(int *a[MAXROWS], int m, int n)
/* read in a table of integers */

{
    int row, col;

    for (row = 0; row < m; ++row) {
        printf("\nEnter data for row no. %d\n", row + 1);
        for (col = 0; col < n; ++col)
            scanf("%d", (*(a + row) + col));
    }
    return;
}

void computesums(int *a[MAXROWS], int *b[MAXROWS], int *c[MAXROWS], int m, int n)
/* add the elements of two integer tables */

{
    int row, col;

    for (row = 0; row < m; ++row)
        for (col = 0; col < n; ++col)
            *(*(c + row) + col) = *(*(a + row) + col) + *(*(b + row) + col);
    return;
}

void writeoutput(int *a[MAXROWS], int m, int n)
/* write out a table of integers */

{
    int row, col;

    for (row = 0; row < m; ++row) {
        for (col = 0; col < n; ++col)
            printf("%4d", *(*(a + row) + col));
        printf("\n");
    }
    return;
}

```

In this program **a**, **b** and **c** are each defined as an array of pointers to integers. Each array has a maximum of **MAXROWS** elements. The function prototypes and the formal argument declarations within the subordinate functions also represent the arrays in this manner.

Since each element of **a**, **b** and **c** is a pointer, we must provide each pointer with enough memory for each row of integer quantities, using the **malloc** function as described in Sec. 10.5. These memory allocations appear in **main**, after the values for **nrows** and **ncols** have been entered. Consider the first memory allocation; i.e.,

```
a[row] = (int *) malloc(ncols * sizeof(int));
```

In this statement **a[0]** points to the the first row. Similarly, **a[1]** points to the second row, **a[2]** points to the third row, and so on. Thus, each array element points to a block of memory large enough to store one row of integer quantities (**ncols** integer quantities). Similar memory allocations are written for the other two arrays.

The individual array elements are processed by repeated use of the indirection operator. In **readininput**, for example, each array element is referenced as

```
scanf("%d", (*(a + row) + col));
```

Similarly, the addition of the array elements within `computesums` is written as

```
*(*(c + row) + col) = *(*(a + row) + col) + *(*(b + row) + col);
```

and the first `printf` statement within `writeoutput` is written as

```
printf("%4d", *(*(a + row) + col));
```

We could, of course, have used the more conventional notation within the functions. Thus, in `readinput` we could have written

```
scanf("%d", a[row][col]);
```

instead of

```
scanf("%d", (*(a + row) + col));
```

Similarly, in `computesums` we could have written

```
c[row][col] = a[row][col] + b[row][col];
```

instead of

```
*(*(c + row) + col) = *(*(a + row) + col) + *(*(b + row) + col);
```

and in `writeoutput` we could have written

```
printf("%4d", a[row][col]);
```

rather than

```
printf("%4d", *(*(a + row) + col));
```

This program will generate output identical to that shown in Example 9.19 when executed with the same input data.

10.8 ARRAYS OF POINTERS

A multidimensional array can be expressed in terms of an array of pointers rather than a pointer to a group of contiguous arrays. In such situations the newly defined array will have one less dimension than the original multidimensional array. Each pointer will indicate the beginning of a separate $(n - 1)$ -dimensional array.

In general terms, a two-dimensional array can be defined as a one-dimensional array of pointers by writing

```
data-type *array[expression 1];
```

rather than the conventional array definition,

```
data-type array[expression 1][expression 2];
```

Similarly, an n -dimensional array can be defined as an $(n - 1)$ -dimensional array of pointers by writing

```
data-type *array[expression 1][expression 2] . . . [expression n-1];
```

rather than

```
data-type array[expression 1][expression 2] . . . [expression n];
```

In these declarations *data-type* refers to the data type of the original *n*-dimensional array, *array* is the array name, and *expression 1*, *expression 2*, . . . , *expression n* are positive-valued integer expressions that indicate the maximum number of elements associated with each subscript.

Notice that the array name and its preceding asterisk are *not* enclosed in parentheses in this type of declaration. (Compare carefully with the pointer declarations presented in the last section.) Thus, a right-to-left rule first associates the pairs of square brackets with *array*, defining the named object as an array. The preceding asterisk then establishes that the array will contain pointers.

Moreover, note that the *last* (i.e., the rightmost) expression is omitted when defining an array of pointers, whereas the *first* (i.e., the leftmost) expression is omitted when defining a pointer to a group of arrays. (Again, compare carefully with the declarations presented in the last section.) You should understand the distinction between this type of declaration and that presented in the last section.

When an *n*-dimensional array is expressed in this manner, an individual array element within the *n*-dimensional array can be accessed by a single use of the indirection operator. The following example illustrates how this is done.

EXAMPLE 10.23 Suppose *x* is a two-dimensional integer array having 10 rows and 20 columns, as in Example 10.20. We can define *x* as a one-dimensional array of pointers by writing

```
int *x[10];
```

Hence, *x[0]* points to the beginning of the first row, *x[1]* points to the beginning of the second row, and so on. Note that the number of elements within each row is not explicitly specified.

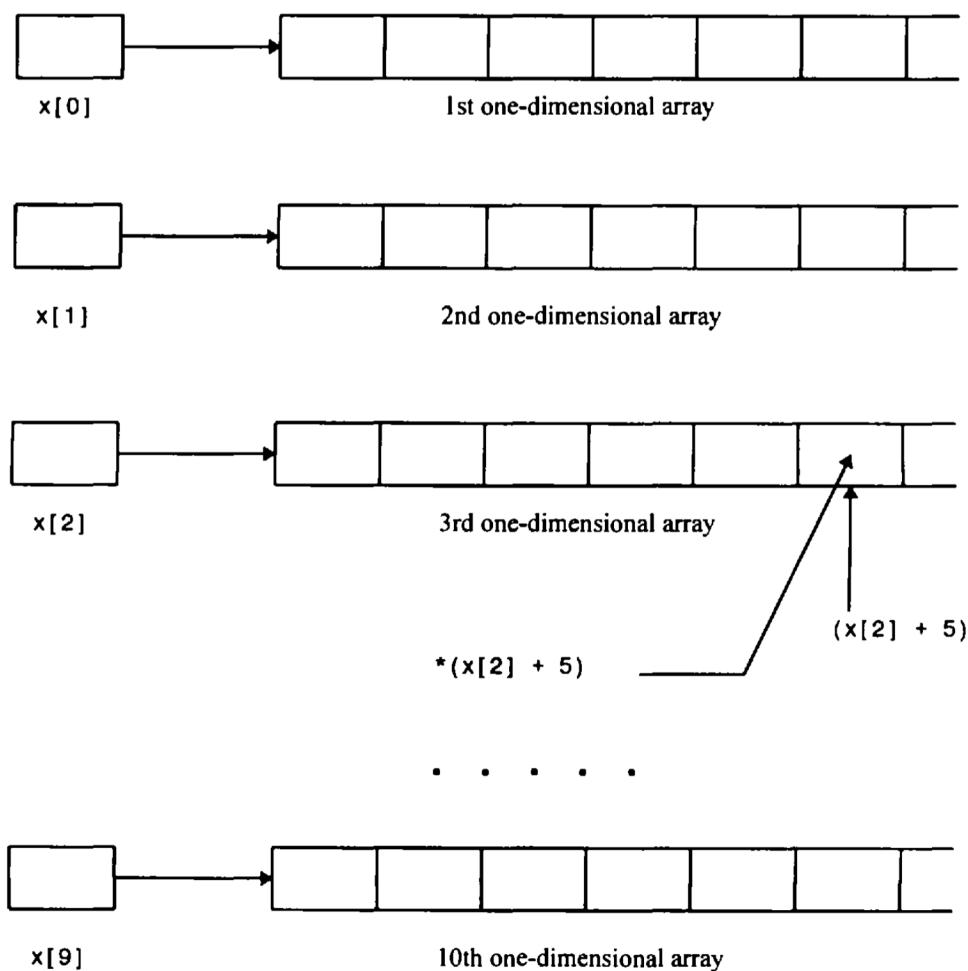


Fig. 10.6

An individual array element, such as $x[2][5]$, can be accessed by writing

```
*(x[2] + 5)
```

In this expression, $x[2]$ is a pointer to the first element in row 2, so that $(x[2] + 5)$ points to element 5 (actually, the sixth element) within row 2. The object of this pointer, $*(x[2] + 5)$, therefore refers to $x[2][5]$. These relationships are illustrated in Fig. 10.6.

Now consider a three-dimensional floating-point array t . Suppose the dimensionality of t is $10 \times 20 \times 30$. This array can be expressed as a two-dimensional array of pointers by writing

```
float *t[10][20];
```

Therefore we have 200 pointers (10 rows, 20 columns), each pointing to a one-dimensional array.

An individual array element, such as $t[2][3][5]$, can be accessed by writing

```
*(t[2][3] + 5)
```

In this expression, $t[2][3]$ is a pointer to the first element in the one-dimensional array represented by $t[2][3]$. Hence, $(t[2][3] + 5)$ points to element 5 (the sixth element) within this array. The object of this pointer, $*(t[2][3] + 5)$, therefore represents $t[2][3][5]$. This situation is directly analogous to the two-dimensional case described above.

EXAMPLE 10.24 Adding Two Tables of Numbers Here is yet another version of the programs presented in Examples 9.19 and 10.22, which calculate the sum of the corresponding elements of two tables of integers. Now each two-dimensional array is represented as an array of pointers to one-dimensional arrays. Each one-dimensional array will correspond to one row within the original two-dimensional array.

```
/* calculate the sum of the elements in two tables of integers */
/* each 2-dimensional array is represented as an array of pointers
   each pointer indicates a row in the original 2-dimensional array */

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

#define MAXROWS 20
#define MAXCOLS 30

/* function prototypes */
void readinput(int *a[MAXROWS], int nrows, int ncols);
void computesums(int *a[MAXROWS], int *b[MAXROWS],
                  int *c[MAXROWS], int nrows, int ncols);
void writeoutput(int *c[MAXROWS], int nrows, int ncols);

main()
{
    int row, nrows, ncols;
    /* array definitions */
    int *a[MAXROWS], *b[MAXROWS], *c[MAXROWS];
    printf("How many rows? ");
    scanf("%d", &nrows);
    printf("How many columns? ");
    scanf("%d", &ncols);
```

```
/* allocate initial memory */
for (row = 0; row <= nrows; row++) {
    a[row] = (int *) malloc(ncols * sizeof(int));
    b[row] = (int *) malloc(ncols * sizeof(int));
    c[row] = (int *) malloc(ncols * sizeof(int));
}

printf("\n\nFirst table:\n");
readinput(a, nrows, ncols);

printf("\n\nSecond table:\n");
readinput(b, nrows, ncols);

computesums(a, b, c, nrows, ncols);

printf("\n\nSums of the elements:\n\n");
writeoutput(c, nrows, ncols);
}

void readinput(int *a[MAXROWS], int m, int n)
/* read in a table of integers */

{
    int row, col;

    for (row = 0; row < m; ++row) {
        printf("\nEnter data for row no. %2d\n", row + 1);
        for (col = 0; col < n; ++col)
            scanf("%d", (a[row] + col));
    }
    return;
}

void computesums(int *a[MAXROWS], int *b[MAXROWS],
                 int *c[MAXROWS], int m, int n)
/* add the elements of two integer tables */

{
    int row, col;

    for (row = 0; row < m; ++row)
        for (col = 0; col < n; ++col)
            *(c[row] + col) = *(a[row] + col) + *(b[row] + col);
    return;
}

void writeoutput(int *a[MAXROWS], int m, int n)
/* write out a table of integers */

{
    int row, col;

    for (row = 0; row < m; ++row) {
        for (col = 0; col < n; ++col)
            printf("%4d", *(a[row] + col));
        printf("\n");
    }
    return;
}
```

Notice that `a`, `b` and `c` are now defined as one-dimensional arrays of pointers. Each array will contain `MAXROWS` elements (i.e., `MAXROWS` pointers). Each array element will point to a one-dimensional array of integers. The function prototypes and the formal argument declarations within the subordinate functions also represent the arrays in this manner.

Each one-dimensional array that is the object of a pointer (i.e., each row within each of the tables) must be allocated an initial block of memory. The `malloc` function accomplishes this. For example, each row within the first table is allocated an initial block of memory in the following manner.

```
a[row] = (int *) malloc(ncols * sizeof(int));
```

This statement associates a block of memory large enough to store `ncols` integer quantities with each pointer (i.e., with each element of `a`). Similar memory allocations are written for `b` and `c`. These `malloc` statements are placed within a `for` loop in order to allocate a block of memory for each of the nonzero rows within the three tables.

Notice the way the individual array elements are processed, using a combination of array and pointer notation. For example, in `readinput` each array element is now referenced as

```
scanf("%d", *(a[row] + col));
```

Similarly, in `computesums` and `writeoutput` the individual array elements are referenced as

```
*(c[row] + col) = *(a[row] + col) + *(b[row] + col);
```

and

```
printf("%4d", *(a[row] + col));
```

respectively. These statements could also have been written using conventional two-dimensional array notation.

This program, as well as the program presented in Example 10.22, will generate output that is identical to that shown in Example 9.19 when executed with the same input data. You may wish to verify this on your own. If this problem were being programmed from scratch, however, the conventional approach shown in Example 9.19, using two-dimensional arrays, would most likely be chosen.

Pointer arrays offer a particularly convenient method for storing strings. In this situation, each array element is a character-type pointer that indicates the beginning of a separate string. Thus, an n -element array can point to n different strings. Each individual string can be accessed by referring to its corresponding pointer.

EXAMPLE 10.25 Suppose the following strings are to be stored in a character-type array.

```
PACIFIC
ATLANTIC
INDIAN
CARIBBEAN
BERING
BLACK
RED
NORTH
BALTIC
CASPIAN
```

These strings can be stored in a two-dimensional character-type array; e.g.,

```
char names[10][12];
```

Note that `names` contains 10 rows, to accommodate the 10 strings. Each row must be large enough to store at least 10 characters, since `CARIBBEAN` contains 9 letters plus the null character (`\0`) at the end. To provide for larger strings, we are allowing each row to contain as many as 12 characters.

A better way to do this is to define a 10-element array of pointers; i.e.

```
char *names[10];
```

Thus, `names[0]` will point to `PACIFIC`, `names[1]` will point to `ATLANTIC`, and so on. Note that it is not necessary to include a maximum string size within the array declaration. However, a specified amount of memory will have to be allocated for each string later in the program, e.g.,

```
names[i] = (char *) malloc(12 * sizeof(char));
```

Just as individual strings can be accessed by referring to the corresponding pointer (i.e., the corresponding array element), so can individual string elements be accessed through the use of the indirection operator. For example, `*(*(names + 2) + 3)` refers to the fourth character (i.e., character number 3) in the third string (row number 2) of the array `names`, as defined in the preceding example.

Rearrangement of the strings can be accomplished simply by reassigning the pointers (i.e., by reassigning the elements in an array of pointers). The strings themselves need not be moved.

EXAMPLE 10.26 Reordering a List of Strings Consider once again the problem of entering a list of strings into the computer and rearranging them into alphabetical order. We saw one approach to this problem in Example 9.20, where the list of strings was stored in a two-dimensional array. Let us now approach this problem using a one-dimensional array of pointers, where each pointer indicates the beginning of a string. The string interchanges can now be carried out simply by reassigning the pointers, as required.

The complete program is presented below.

```
/* sort a list of strings into alphabetical order using an array of pointers */

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

void reorder(int n, char *x[]);

main()
{
    int i, n = 0;
    char *x[10];

    printf("Enter each string on a separate line below\n\n");
    printf("Type '\\END\\' when finished\n\n");

    /* read in the list of strings */
    do {
        /* allocate memory */
        x[n] = (char *) malloc(12 * sizeof(char));

        printf("string %d: ", n + 1);
        scanf("%s", x[n]);
    }
    while (strcmp(x[n++], "\\END\\"));

    /* reorder the list of strings */
    reorder(--n, x);

    /* display the reordered list of strings */
    printf("\n\nReordered List of Strings:\n");
    for (i = 0; i < n; ++i)
        printf("\nstring %d: %s", i + 1, x[i]);
}
```

```

void reorder(int n, char *x[]) /* rearrange the list of strings */

{
    char *temp;
    int i, item;

    for (item = 0; item < n - 1; ++item)
        /* find the lowest of all remaining strings */
        for (i = item + 1; i < n; ++i)

            if (strcmp(x[item], x[i]) > 0) {
                /* interchange the two strings */
                temp = x[item];
                x[item] = x[i];
                x[i] = temp;
            }
    return;
}

```

The logic is essentially the same as that shown in Example 9.20, though the array containing the strings is now defined as an array of pointers. Notice that the second formal argument in the function `reorder` is declared in the same manner. Also, notice the string interchange routine (i.e., the `if` statement) within `reorder`. It is now the pointers, not the actual strings, that are interchanged. Hence the library function `strcpy`, which was used in Example 9.20, is not required. The program will therefore run somewhat faster than the earlier version.

Execution of this program will generate the same dialog as that shown in Example 9.20.

If the elements of an array are string pointers, a set of initial values can be specified as a part of the array declaration. In such cases the initial values will be strings, where each string corresponds to a separate array element. Remember, however, that an array must be declared `static` if it is initialized within a function.

An advantage to this scheme is that a fixed block of memory need not be reserved in advance, as is done when initializing a conventional array. Thus, if the initial declaration includes many strings and some of them are relatively short, there may be a substantial savings in memory allocation. Moreover, if some of the strings are particularly long, there is no need to worry about the possibility of exceeding some maximum specified string length (i.e., the maximum number of characters per row). Arrays of this type are often referred to as *ragged arrays*.

EXAMPLE 10.27 The following array declaration appears within a function.

```

static char *names[10] = {
    "PACIFIC",
    "ATLANTIC",
    "INDIAN",
    "CARIBBEAN",
    "BERING",
    "BLACK",
    "RED",
    "NORTH",
    "BALTIc",
    "CASPIAN"
};

```

In this example, `names` is a 10-element array of pointers. Thus, the first array element (i.e., the first pointer) will point to `PACIFIC`, the second array element will point to `ATLANTIC`, and so on.

Notice that the array is declared as **static** so that it can be initialized within the function. If the array declaration were external to all program functions, the **static** storage class designation would not be necessary.

Since the array declaration includes initial values, it is really not necessary to include an explicit size designation within the declaration. The size of the array will automatically be set equal to the number of strings that are present. Thus, the above declaration can be written as

```
static char *names[] = {
    "PACIFIC",
    "ATLANTIC",
    "INDIAN",
    "CARIBBEAN",
    "BERING",
    "BLACK",
    "RED",
    "NORTH",
    "BALTIC",
    "CASPIAN"
};
```

It should be understood that the ragged-array concept refers only to the *initialization* of string arrays, not the assignment of individual strings that may be read into the computer via the **scanf** function. Thus, applications requiring that strings be read into the computer and then processed, as in Example 10.26, still require the allocation of a specified amount of memory for each array element.

Initialized string values can be accessed by referring to their corresponding pointers (i.e., their corresponding array elements), in the usual manner. These pointers can be reassigned other string constants elsewhere in the program if necessary.

EXAMPLE 10.28 Displaying the Day of the Year Let us develop a program that will accept three integer quantities, indicating the month, day and year, and then display the corresponding day of the week, the month, the day and the year in a more legible manner. For example, suppose we were to enter the date 5 24 1997; this would produce the output **Saturday, May 24, 1997**. Programs of this type are often used to display information that is stored in a computer's internal memory in an encoded format.

Our basic strategy will be to enter a date into the computer, in the form *month, day, year* (**mm dd yyyy**), and then convert this date into the number of days relative to some base date. The day of the week corresponding to the specified date can then be determined quite easily, provided we know the day of the week corresponding to the base date. Let us arbitrarily choose Monday, January 1, 1900 as the base date. We will then convert any date beyond January 1, 1900 (actually, any date between January 1, 1900 and December 31, 2099) into an equivalent day of the week.

The computation can be carried out using the following empirical rules.

1. Enter numerical values for the variables **mm**, **dd** and **yy**, which represent the month, day and year, respectively (e.g., 5 24 1997).
2. Determine the approximate day of the current year, as


```
ndays = (long) (30.42 * (mm - 1)) + dd;
```
3. If **mm == 2** (February), increase the value of **ndays** by 1.
4. If **mm > 2** and **mm < 8** (March, April, May, June or July), decrease the value of **ndays** by 1.
5. Convert the year into the number of years beyond the base date; i.e., **yy -= 1900**. Then test for a leap year as follows: If **(yy % 4) == 0** and **mm > 2**, increase the value of **ndays** by 1.
6. Determine the number of complete 4-year cycles beyond the base date as **yy / 4**. For each complete 4-year cycle, add 1461 to **ndays**.
7. Determine the number of full years beyond the last complete 4-year cycle as **yy % 4**. For each full year, add 365 to **ndays**. Then add 1, because the first year beyond a full 4-year cycle will be a leap year.

8. If `ndays > 59` (i.e., if the date is any day beyond February 28, 1900), decrease the value of `ndays` by 1, because 1900 is not a leap year. (Note that the last year of each century is not a leap year, except those years that are evenly divisible by 400. Therefore 1900, the last year of the nineteenth century, is *not* a leap year, but 2000, the last year of the twentieth century, *is* a leap year.)

9. Determine the numerical day of the week corresponding to the specified date as `day = (ndays % 7)`.

Note that `day == 1` corresponds either to the base date, which is a Monday, or another date that also occurs on a Monday. Hence, `day == 2` will refer to a Tuesday, `day == 3` will refer to a Wednesday, . . . , `day == 6` will refer to a Saturday, and `day == 0` will refer to a Sunday.

Here is a complete function, called `convert`, that carries out steps 2 through 9. Note that `convert` accepts the integers `mm`, `dd` and `yy` as input parameters, and returns the integer quantity `(ndays % 7)`. Also, notice that `ndays` and `ncycles` are long integer variables, whereas all other variables are ordinary integers.

```
int convert(int mm, int dd, int yy) /* convert date to numerical day of week */

{
    long ndays;           /* number of days from start of 1900 */
    long ncycles;         /* number of 4-year cycles beyond 1900 */
    int nyears;           /* number of years beyond last 4-year cycle */
    int day;              /* day of week (0, 1, 2, 3, 4, 5 or 6) */

    /* numerical conversions */
    yy -= 1900;
    ndays = (long) (30.42 * (mm - 1)) + dd; /* approximate day of year */

    if (mm == 2) ++ndays;                      /* adjust for February */
    if ((mm > 2) && (mm < 8)) --ndays;        /* adjust for March - July */
    if ((yy % 4 == 0) && (mm > 2)) ++ndays;    /* adjust for leap year */

    ncycles = yy / 4;                         /* 4-year cycles beyond 1900 */
    ndays += ncycles * 1461;                   /* add days for 4-year cycles */

    nyears = yy % 4;                          /* years beyond last 4-year cycle */
    if (nyears > 0)                          /* add days for yrs beyond last 4-year cycle */
        ndays += 365 * nyears + 1;

    if (ndays > 59) --ndays;                  /* adjust for 1900 (NOT a leap year) */
    day = ndays % 7;

    return(day);
}
```

The names of the days of the week can be stored as strings in a 7-element array; i.e.,

```
static char *weekday[] = {"Sunday", "Monday", "Tuesday", "Wednesday",
                         "Thursday", "Friday", "Saturday"};
```

Each day corresponds to the value assigned to `day`, where `day = (ndays % 7)`. The days begin with `Sunday` because `Sunday` corresponds to `day == 0`, as explained above. If the base date were not a Monday, this particular ordering of the days of the week would have to be changed.

Similarly, the names of the months can be stored as strings in a 12-element array; i.e.,

```
static char *month[] = {"January", "February", "March", "April", "May", "June", "July",
                       "August", "September", "October", "November", "December"};
```

Each month corresponds to the value of `mm - 1`.

Here is an entire C program that will carry out the conversion interactively.

```
/* convert a numerical date (mm dd yyyy) into 'day of week, month, day, year'
   (e.g., 5 24 1997 -> Saturday, May 24, 1997) */

#include <stdio.h>

void readinput(int *pm, int *pd, int *py);      /* function prototype */
int convert(int mm, int dd, int yy);            /* function prototype */

main()
{
    int mm, dd, yy;
    int day_of_week;    /* day of the week (0 -> Sunday,
                           1 -> Monday,
                           . . .
                           6 -> Saturday) */

    static char *weekday[] = {"Sunday", "Monday", "Tuesday", "Wednesday",
                             "Thursday", "Friday", "Saturday"};

    static char *month[] = {"January", "February", "March", "April",
                           "May", "June", "July", "August", "September",
                           "October", "November", "December"};

    /* opening message */
    printf("Date Conversion Routine\nTo STOP, enter 0 0 0");

    readinput(&mm, &dd, &yy);

    /* convert date to numerical day of week */
    while (mm > 0)  {
        day_of_week = convert(mm, dd, yy);
        printf("\n%s, %s %d, %d", weekday[day_of_week], month[mm-1], dd, yy);

        readinput(&mm, &dd, &yy);
    }
}

void readinput(int *pm, int *pd, int *py) /* read in the numerical date */
{
    printf("\n\nEnter mm dd yyyy: ");
    scanf("%d %d %d", pm, pd, py);
    return;
}

int convert(int mm, int dd, int yy) /* convert date to numerical day of week */
{
    long ndays;          /* number of days from start of 1900 */
    long ncycles;        /* number of 4-year cycles beyond 1900 */
    int nyears;          /* number of years beyond last 4-year cycle */
    int day;              /* day of week (0, 1, 2, 3, 4, 5 or 6) */
```

```

/* numerical conversions */
yy -= 1900;
ndays = (long) (30.42 * (mm - 1)) + dd; /* approximate day of year */

if (mm == 2) ++ndays; /* adjust for February */
if ((mm > 2) && (mm < 8)) --ndays; /* adjust for March - July */
if ((yy % 4 == 0) && (mm > 2)) ++ndays; /* adjust for leap year */

ncycles = yy / 4; /* 4-year cycles beyond 1900 */
ndays += ncycles * 1461; /* add days for 4-year cycles */

nyears = yy % 4; /* years beyond last 4-year cycle */
if (nyears > 0) /* add days for yrs beyond last 4-year cycle */
    ndays += 365 * nyears + 1;

if (ndays > 59) --ndays; /* adjust for 1900 (NOT a leap year) */

day = ndays % 7;

return(day);
}

```

This program includes a loop that repeatedly accepts a date in the form of three integers (i.e., mm dd yyyy) and returns the corresponding day and date in a more legible form. The program will continue to run until a value of 0 is entered for mm. Note that the prompt indicates that three zeros must be entered in order to stop the program execution; i.e., 0 0 0. Actually, however, the program only checks the value of mm.

A typical interactive session is shown below. As usual, the user's responses are underlined.

Date Conversion Routine
To STOP, enter 0 0 0

Enter mm dd yyyy: 10 29 1929

Tuesday, October 29, 1929

Enter mm dd yyyy: 8 15 1945

Wednesday, August 15, 1945

Enter mm dd yyyy: 7 20 1969

Sunday, July 20, 1969

Enter mm dd yyyy: 5 24 1997

Saturday, May 24, 1997

Enter mm dd yyyy: 8 30 2010

Monday, August 30, 2010

Enter mm dd yyyy: 4 12 2069

Friday, April 12, 2069

Enter mm dd yyyy: 0 0 0

10.9 PASSING FUNCTIONS TO OTHER FUNCTIONS

A pointer to a function can be passed to another function as an argument. This allows one function to be transferred to another, as though the first function were a variable. Let us refer to the first function as the *guest function*, and the second function as the *host function*. Thus, the guest is passed to the host, where it can be accessed. Successive calls to the host function can pass different pointers (i.e., different guest functions) to the host.

When a host function accepts the name of a guest function as an argument, the formal argument declaration must identify that argument as a pointer to the guest function. In its simplest form, the formal argument declaration can be written as

data-type (**function-name*)()

where *data-type* refers to the data type of the quantity returned by the guest and *function-name* is the name of the guest. The formal argument declaration can also be written as

data-type (**function-name*)(*type 1*, *type 2*, . . .)

or as

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

where *type 1*, *type 2*, . . . refer to the data types of the arguments associated with the guest, and *arg 1*, *arg 2*, . . . refer to the names of the arguments associated with the guest.

The guest function can be accessed within the host by means of the indirection operator. To do so, the indirection operator must precede the guest function name (i.e., the formal argument). Both the indirection operator and the guest function name must be enclosed in parentheses; i.e.,

(*function-name)(argument 1, argument 2, . . ., argument n);

where *argument 1*, *argument 2*, . . . , *argument n* refer to the arguments that are required in the function call.

Now consider the function declaration for the host function. It may be written as

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

← pointer to guest function →|

data types of other funct args);

where *funct-data-type* refers to the data type of the quantity returned by the host function; *funct-name* refers to the name of the host function; *arg-data-type* refers to the data type of the quantity returned by the guest function, and *type 1*, *type 2*, . . . refer to the data types of guest function's arguments. Notice that the indirection operator appears in parentheses, to indicate a pointer to the guest function. Moreover, the data types of the guest function's arguments follow in a separate pair of parentheses, to indicate that they are function arguments.

When full function prototyping is used, the host function declaration is expanded as follows.

funct-data-type *funct-name*

(arg-data-type (*pt-var)(type 1 arg 1, type 2 arg 2, . . .),

1

pointer to guest function

—→

data types and names of other funct args);

The notation is the same as above, except that *pt-var* refers to the pointer variable pointing to the guest function, and *type 1 arg 1, type 2 arg 2, . . .* refer to the data types and the corresponding names of the guest function's arguments.

EXAMPLE 10.29 The skeletal outline of a C program is shown below. This program consists of four functions: **main**, **process**, **funct1** and **funct2**. Note that **process** is a host function for **funct1** and **funct2**. Each of the three subordinate functions returns an integer quantity.

```

int process(int (*)(int, int));      /* function declaration (host) */
int funct1(int, int);              /* function declaration (guest) */
int funct2(int, int);              /* function declaration (guest) */

main()
{
    int i, j;
    . . . .
    i = process(funct1); /* pass funct1 to process; return a value for i */
    . . . .
    j = process(funct2); /* pass funct2 to process; return a value for j */
    . . . .
}

process(int (*pf)(int, int))          /* host function definition */
                                         /* (formal argument is a pointer to a function) */
{
    int a, b, c;
    . . . .
    c = (*pf)(a, b); /* access the function passed to this function;
                        return a value for c */
    . . . .
    return(c);
}

funct1(a, b)           /* guest function definition */
{
    int a, b;
    {
        int c;
        c = . . . /* use a and b to evaluate c */
        return(c);
    }
}

funct2(x, y)           /* guest function definition */
{
    int x, y;
    {
        int z;
        z = . . . /* use x and y to evaluate z */
        return(z);
    }
}

```

Notice that this program contains three function declarations. The declarations for funct1 and funct2 are straightforward. However the declaration for process requires some explanation. This declaration states that process is a host function that returns an integer quantity and has one argument. The argument is a pointer to a guest function that returns an integer quantity and has two integer arguments. The argument designation for the guest function is written as

```
int (*)(int, int)
```

Notice the way the argument designation fits into the entire host function declaration; i.e.,

```
int process(int (*)(int, int));
```

Now consider the formal argument declaration that appears within process; i.e.,

```
int (*pf)(int, int);
```

This declaration states that pf is a pointer to a guest function. The guest function will return an integer quantity, and it requires two integer arguments.

Here is another version of this same outline, utilizing full function prototyping. The changes are shown in boldface.

```
int process(int (*pf)(int a, int b)); /* function prototype (host) */
int funct1(int a, int b);           /* function prototype (guest) */
int funct2(int a, int b);           /* function prototype (guest) */

main()
{
    int i, j;
    . . .
    i = process(funct1); /* pass funct1 to process; return a value for i */
    . . .
    j = process(funct2); /* pass funct2 to process; return a value for j */
    . . .

}

process(int (*pf)(int a, int b)) /* host function definition */
{
    int a, b, c;
    . . .
    c = (*pf)(a, b); /* access the function passed to this function;
                        return a value for c */
    . . .
    return(c);
}

funct1(int a, int b) /* guest function definition */
{
    int c;
    c = . . .          /* use a and b to evaluate c */
    return(c);
}
```

```

funct2(int x, int y)      /* guest function definition */
{
    int z;

    z = . . .           /* use x and y to evaluate z */

    return(z);
}

```

The function prototypes include argument names as well as argument data types. Moreover, the prototype for **process** now includes the name of the variable (**pf**) that points to the guest function. Notice that the declaration of the formal argument **pf** within **process** is consistent with the function prototype.

Some programming applications can be formulated quite naturally in terms of one function being passed to another. For example, one function might represent a mathematical equation, and the other might contain a computational strategy to process the equation. In such cases the function representing the equation might be passed to the function that processes the equation. This is particularly useful if the program contains several different mathematical equations, one of which is selected by the user each time the program is executed.

EXAMPLE 10.30 Future Value of Monthly Deposits (Compound Interest Calculations) Suppose a person decides to save a fixed amount of money at the end of every month for n years. If the money earns interest at i percent per year, then it is natural to ask how much money will accumulate after n years (i.e., after $12n$ monthly deposits). The answer, of course, depends upon how much money is deposited each month, the interest rate, and the frequency with which the interest is compounded. For example, if the interest is compounded annually, semiannually, quarterly or monthly, the future amount of money that will accumulate after n years is given by

$$F = \frac{12A}{m} \left[\frac{(1+i/m)^{mn} - 1}{i/m} \right] = 12A \left[\frac{(1+i/m)^{mn} - 1}{i} \right]$$

where F is the future accumulation, A is the amount of money deposited each month, i is the annual interest rate (expressed as a decimal), and m is the number of compounding periods per year (e.g., $m = 1$ for annual compounding, $m = 2$ for semiannual compounding, $m = 4$ for quarterly compounding and $m = 12$ for monthly compounding).

If the compounding periods are shorter than the payment periods, such as in the case of daily compounding, then the future amount of money is determined by

$$F = A \left[\frac{(1+i/m)^{mn} - 1}{(1+i/m)^{m/12} - 1} \right]$$

Note that m is customarily assigned a value of 360 when the interest is compounded daily.

Finally, in the case of continuous compounding, the future amount of money is determined as

$$F = A \left[\frac{e^{in} - 1}{e^{i/12} - 1} \right]$$

Suppose we wish to determine F as a function of the annual interest rate i , for given values of A , m and n . Let us develop a program that will read the required input data into **main**, and then carry out the calculations within a separate function, called **table**. Each of the three formulas for determining the ratio F/A will be placed in one of three independent functions, called **md1**, **md2** and **md3**, respectively. Thus, the program will consist of five different functions.

When **table** is called from **main**, one of the arguments passed to **table** will be the name of the function containing the appropriate formula, as indicated by an input parameter (**freq**). The values of A , m and n that are read into **main** will also be passed to **table** as arguments. A loop will then be initiated within **table**, in which values of F are determined

for interest rates ranging from 0.01 (i.e., 1 percent per year) to 0.20 (20 percent per year). The calculated values will be displayed as they are generated. The entire program is shown below.

```
/* personal finance calculations */

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

/* function prototypes */
void table (double (*pf)(double i, int m, double n), double a, int m, double n);
double md1(double i, int m, double n);
double md2(double i, int m, double n);
double md3(double i, int m, double n);

main()      /* calculate the future value of a series of monthly deposits */
{
    int m;          /* number of compounding periods per year */
    double n;        /* number of years */
    double a;        /* amount of each monthly payment */
    char freq;       /* frequency of compounding indicator */

    /* enter input data */
    printf("\nFUTURE VALUE OF A SERIES OF MONTHLY DEPOSITS\n\n");
    printf("Amount of Each Monthly Payment: ");
    scanf("%lf", &a);
    printf("Number of Years: ");
    scanf("%lf", &n);

    /* enter frequency of compounding */
    do   {
        printf("Frequency of Compounding (A, S, Q, M, D, C): ");
        scanf("%1s", &freq);
        freq = toupper(freq);      /* convert to upper case */
        if (freq == 'A')    {
            m = 1;
            printf("\nAnnual Compounding\n");
        }
        else if (freq == 'S')   {
            m = 2;
            printf("\nSemiannual Compounding\n");
        }
        else if (freq == 'Q')   {
            m = 4;
            printf("\nQuarterly Compounding\n");
        }
        else if (freq == 'M')   {
            m = 12;
            printf("\nMonthly Compounding\n");
        }
        else if (freq == 'D')   {
            m = 360;
            printf("\nDaily Compounding\n");
        }
    }
```

```

        else if (freq == 'C')    {
            m = 0;
            printf("\nContinuous Compounding\n");
        }
        else
            printf("\nERROR - Please Repeat\n\n");
    } while (freq != 'A' && freq != 'S' && freq != 'Q' &&
           freq != 'M' && freq != 'D' && freq != 'C');

/* carry out the calculations */
if (freq == 'C')
    table(md3, a, m, n); /* continuous compounding */
else if (freq == 'D')
    table(md2, a, m, n); /* daily compounding */
else
    table(md1, a, m, n); /* annual, semiannual, quarterly or monthly compounding */
}

void table (double (*pf)(double i, int m, double n), double a, int m, double n)
/* table generator (this function accepts a pointer to another function as an argument)

NOTE: double (*pf)(double i, int m, double n) is a POINTER TO A FUNCTION */

{
    int count; /* loop counter */
    double i; /* annual interest rate */
    double f; /* future value */

    printf("\nInterest Rate      Future Amount\n\n");
    for (count = 1; count <= 20; ++count) {
        i = 0.01 * count;
        f = a * (*pf)(i, m, n); /* ACCESS THE FUNCTION PASSED AS A POINTER */
        printf("      %2d          %.2f\n", count, f);
    }
    return;
}

double md1(double i, int m, double n)
/* monthly deposits, periodic compounding */

{
    double factor, ratio;

    factor = 1 + i/m;
    ratio = 12 * (pow(factor, m*n) - 1) / i;
    return(ratio);
}

double md2(double i, int m, double n)
/* monthly deposits, daily compounding */

{
    double factor, ratio;

    factor = 1 + i/m;
    ratio = (pow(factor, m*n) - 1) / (pow(factor, m/12) - 1);
    return(ratio);
}

```

```

double md3(double i, int dummy, double n)
/* monthly deposits, continuous compounding */

{
    double ratio;

    ratio = (exp(i*n) - 1) / (exp(i/12) - 1);
    return(ratio);
}

```

Notice the function prototypes, particularly the prototype for `table`. The first argument passed to `table` is a pointer to a guest function that receives two double-precision arguments and an integer argument, and returns a double-precision quantity. This pointer is intended to represent `md1`, `md2` or `md3`. The prototypes for these three functions follow the prototype for `table`. Each of these functions accepts two double-precision arguments and an integer argument, and returns a double-precision quantity, as required.

An interactive dialog for the input data is generated within `main`. In particular, the program accepts numerical values for `a` and `n`. It also accepts a one-character string for the character variable `freq`, which indicates the frequency of compounding. The only allowable characters that can be assigned to `freq` are A, S, Q, M, D or C (for Annual, Semiannual, Quarterly, Monthly, Daily or Continuous compounding, respectively). This character can be entered in either upper- or lowercase, since it is converted to uppercase within the program. Note that the program contains an error trap preventing characters other than A, S, Q, M, D or C from being entered.

An appropriate numerical value is assigned to `m` as soon as the frequency of compounding is determined. The program then accesses `table`, passing either `md1`, `md2` or `md3` as an argument, as determined by the character assigned to `freq`. (See the multiple `if - else` statement at the end of `main`.)

Now examine the host function `table`. The last three formal arguments (`a`, `m` and `n`) are declared as ordinary double-precision or integer variables. However, the first formal argument (`pf`) is declared as a pointer to a guest function that accepts two double-precision arguments and an integer argument, and returns a double-precision quantity. These formal argument declarations are consistent with the function prototype for `table`.

The values for `i` (i.e., the interest rates) are generated internally within `table`. These values are determined as 0.01 * count. Since count ranges from 1 to 20, we see that the interest rates range from 0.01 to 0.20, as required.

Notice the manner in which the values for `f` are calculated within `table`; i.e.,

```
f = a * (*pf)(i, m, n);
```

The expression `(*pf)` refers to the guest function whose name is passed to `table` (i.e., either `md1`, `md2` or `md3`). This is accompanied by the required list of arguments, containing the current values for `i`, `m` and `n`. The value returned by the guest function is then multiplied by `a`, and the product is assigned to `f`.

The three remaining functions, `md1`, `md2` and `md3`, are straightforward. Notice that the second argument in `md3` is called `dummy`, because the value of this argument is not utilized within the function. We could have done this with `md2` as well, since the value of `m` is always 360 in the case of daily compounding.

Execution of the program produces the following representative dialog.

FUTURE VALUE OF A SERIES OF MONTHLY DEPOSITS

```

Amount of Each Monthly Payment: 100
Number of Years: 3
Frequency of Compounding (A, S, Q, M, D, C): p

```

```
ERROR - Please Repeat
```

```
Frequency of Compounding (A, S, Q, M, D, C): m
```

```
Monthly Compounding
```

| Interest Rate | Future Amount |
|---------------|---------------|
| 1 | 3653.00 |
| 2 | 3707.01 |
| 3 | 3762.06 |
| 4 | 3818.16 |
| 5 | 3875.33 |
| 6 | 3933.61 |
| 7 | 3993.01 |
| 8 | 4053.56 |
| 9 | 4115.27 |
| 10 | 4178.18 |
| 11 | 4242.31 |
| 12 | 4307.69 |
| 13 | 4374.33 |
| 14 | 4442.28 |
| 15 | 4511.55 |
| 16 | 4582.17 |
| 17 | 4654.18 |
| 18 | 4727.60 |
| 19 | 4802.45 |
| 20 | 4878.78 |

10.10 MORE ABOUT POINTER DECLARATIONS

Before leaving this chapter we mention that pointer declarations can become complicated, and some care is required in their interpretation. This is especially true of declarations that involve functions or arrays.

One difficulty is the dual use of parentheses. In particular, parentheses are used to indicate functions, and they are used for nesting purposes (to establish precedence) within more complicated declarations. Thus, the declaration

```
int *p(int a);
```

indicates a function that accepts an integer argument, and returns a pointer to an integer. On the other hand, the declaration

```
int (*p)(int a);
```

indicates a *pointer to a function* that accepts an integer argument and returns an integer. In this last declaration, the first pair of parentheses is used for nesting, and the second pair is used to indicate a function.

The interpretation of more complex declarations can be increasingly troublesome. For example, consider the declaration

```
int *(*p)(int (*a)[]);
```

In this declaration, `(*p)(. . .)` indicates a pointer to a function. Hence, `int *(*p)(. . .)` indicates a pointer to a function that returns a pointer to an integer. Within the last pair of parentheses (the function's argument specification), `(*a)[]` indicates a pointer to an array. Therefore, `int (*a)[]` represents a pointer to an array of integers. Putting the pieces together, `(*p)(int (*a)[])` represents a pointer to a function whose argument is a pointer to an array of integers. And finally, the entire declaration

```
int *(*p)(int (*a)[]);
```

represents a pointer to a function that accepts a pointer to an array of integers as an argument, and returns a pointer to an integer.

Remember that a left parenthesis immediately following an identifier name indicates that the identifier represents a function. Similarly, a left square bracket immediately following an identifier name indicates that the identifier represents an array. Parentheses that identify functions and square brackets that identify arrays have a higher precedence than the unary indirection operator (see Appendix C). Therefore, additional parentheses are required when declaring a pointer to a function or a pointer to an array.

The following example provides a number of illustrations.

EXAMPLE 10.31 Several declarations involving pointers are shown below. The individual declarations range from simple to complex.

```
int *p;                      /* p is a pointer to an integer quantity */

int *p[10];                  /* p is a 10-element array of pointers to integer quantities */

int (*p)[10];                /* p is a pointer to a 10-element integer array */

int *p(void);                /* p is a function that
                                returns a pointer to an integer quantity */

int p(char *a);              /* p is a function that
                                accepts an argument which is a pointer to a character and
                                returns an integer quantity */

int *p(char a*);             /* p is a function that
                                accepts an argument which is a pointer to a character
                                returns a pointer to an integer quantity */

int (*p)(char *a);            /* p is a pointer to a function that
                                accepts an argument which is a pointer to a character
                                returns an integer quantity */

int (*p(char *a))[10];        /* p is a function that
                                accepts an argument which is a pointer to a character
                                returns a pointer to a 10-element integer array */

int p(char (*a)[]);           /* p is a function that
                                accepts an argument which is a pointer to a character array
                                returns an integer quantity */

int p(char *a[]);             /* p is a function that
                                accepts an argument which is an array of pointers to
                                characters
                                returns an integer quantity */

int *p(char a[]);             /* p is a function that
                                accepts an argument which is a character array
                                returns a pointer to an integer quantity */

int *p(char (*a)[]);          /* p is a function that
                                accepts an argument which is a pointer to a character array
                                returns a pointer to an integer quantity */

int *p(char *a[]);             /* p is a function that
                                accepts an argument which is a character array
                                returns a pointer to an integer quantity */
```

```

        accepts an argument which is an array of pointers to
        characters
        returns a pointer to an integer quantity */

int (*p)(char (*a)[]); /* p is a pointer to a function that
                           accepts an argument which is a pointer to a character array
                           returns an integer quantity */

int *(*p)(char (*a)[]); /* p is pointer to a function that
                           accepts an argument which is a pointer to a character array
                           returns a pointer to an integer quantity */

int *(*p)(char *a[]); /* p is a pointer to a function that
                           accepts an argument which is an array of pointers to
                           characters
                           returns a pointer to an integer quantity */

int (*p[10])(void); /* p is a 10-element array of pointers to functions;
                           each function returns an integer quantity */

int (*p[10])(char a); /* p is a 10-element array of pointers to functions;
                           each function accepts an argument which is a character, and
                           returns an integer quantity */

int *(*p[10])(char a); /* p is a 10-element array of pointers to functions;
                           each function accepts an argument which is a character, and
                           returns a pointer to an integer quantity */

int *(*p[10])(char *a); /* p is a 10-element array of pointers to functions;
                           each function accepts an argument which is a pointer to a
                           character, and
                           returns a pointer to an integer quantity */

```

Review Questions

- 10.1 For the version of C available on your particular computer, how many memory cells are required to store a single character? An integer quantity? A long integer? A floating-point quantity? A double-precision quantity?
- 10.2 What is meant by the address of a memory cell? How are addresses usually numbered?
- 10.3 How is a variable's address determined?
- 10.4 What kind of information is represented by a pointer variable?
- 10.5 What is the relationship between the address of a variable v and the corresponding pointer variable pv?
- 10.6 What is the purpose of the indirection operator? To what type of operand must the indirection operator be applied?
- 10.7 What is the relationship between the data item represented by a variable v and the corresponding pointer variable pv?
- 10.8 What precedence is assigned to the unary operators compared with the multiplication, division and remainder operators? In what order are the unary operators evaluated?
- 10.9 Can the address operator act upon an arithmetic expression, such as $2 * (u + v)$? Explain the reasons for your answer.
- 10.10 Can an expression involving the indirection operator appear on the left side of an assignment statement? Explain.

- 10.11 What kinds of objects can be associated with pointer variables?
- 10.12 How is a pointer variable declared? What is the purpose of the data type included in the declaration?
- 10.13 In what way can the assignment of an initial value be included in the declaration of a pointer variable?
- 10.14 Are integer values ever assigned to pointer variables? Explain.
- 10.15 Why is it sometimes desirable to pass a pointer to a function as an argument?
- 10.16 Suppose a function receives a pointer as an argument. Explain how the function prototype is written. In particular, explain how the data type of the pointer argument is represented.
- 10.17 Suppose a function receives a pointer as an argument. Explain how the pointer argument is declared within the function definition.
- 10.18 What is the relationship between an array name and a pointer? How is an array name interpreted when it appears as an argument to a function?
- 10.19 Suppose a formal argument within a function definition is an array. How can the array be declared within the function?
- 10.20 How can a portion of an array be passed to a function?
- 10.21 How can a function return a pointer to its calling routine?
- 10.22 Describe two different ways to specify the address of an array element.
- 10.23 Why is the value of an array subscript sometimes referred to as an offset when the subscript is a part of an expression indicating the address of an array element?
- 10.24 Describe two different ways to access an array element. Compare your answer to that of question 10.22.
- 10.25 Can an address be assigned to an array name or an array element? Can an address be assigned to a pointer variable whose object is an array?
- 10.26 Suppose a numerical array is defined in terms of a pointer variable. Can the individual array elements be initialized?
- 10.27 Suppose a character-type array is defined in terms of a pointer variable. Can the individual array elements be initialized? Compare your answer with that of the previous question.
- 10.28 What is meant by dynamic memory allocation? What library function is used to allocate memory dynamically? How is the size of the memory block specified? What kind of information is returned by the library function?
- 10.29 Suppose an integer quantity is added to or subtracted from a pointer variable. How will the sum or difference be interpreted?
- 10.30 Under what conditions can one pointer variable be subtracted from another? How will this difference be interpreted?
- 10.31 Under what conditions can two pointer variables be compared? Under what conditions are such comparisons useful?
- 10.32 How is a multidimensional array defined in terms of a pointer to a collection of contiguous arrays of lower dimensionality?
- 10.33 How can the indirection operator be used to access a multidimensional array element?
- 10.34 How is a multidimensional array defined in terms of an array of pointers? What does each pointer represent? How does this definition differ from a pointer to a collection of contiguous arrays of lower dimensionality?
- 10.35 How can a one-dimensional array of pointers be used to represent a collection of strings?
- 10.36 If several strings are stored within a one-dimensional array of pointers, how can an individual string be accessed?
- 10.37 If several strings are stored within a one-dimensional array of pointers, what happens if the strings are reordered? Are the strings actually moved to different locations within the array?
- 10.38 Under what conditions can the elements of a multidimensional array be initialized if the array is defined in terms of an array of pointers?
- 10.39 When transferring one function to another, what is meant by the guest function? What is the host function?

- 10.40** Suppose a formal argument within a host function definition is a pointer to another function. How is the formal argument declared? Within the declaration, to what does the data type refer?
- 10.41** Suppose a formal argument within the definition of the host function p is a pointer to the guest function q. How is the formal argument declared within p? In this declaration, to what does the data type refer? How is function q accessed within function p?
- 10.42** Suppose that p is a host function, and one of p's arguments is a pointer to function q. How would the declaration for p be written if full function prototyping is used?
- 10.43** For what types of applications is it particularly useful to pass one function to another?

Problems

- 10.44** Explain the meaning of each of the following declarations.

- (a) `int *px;`
- (b) `float a, b;`
`float *pa, *pb;`
- (c) `float a = -0.167;`
`float *pa = &a;`
- (d) `char c1, c2, c3;`
`char *pc1, *pc2, *pc3 = &c1;`
- (e) `double funct(double *a, double *b, int *c);`
- (f) `double *funct(double *a, double *b, int *c);`
- (g) `double (*a)[12];`
- (h) `double *a[12];`
- (i) `char *a[12];`
- (j) `char *d[4] = {"north", "south", "east", "west"};`
- (k) `long (*p)[10][20];`
- (l) `long *p[10][20];`
- (m) `char sample(int (*pf)(char a, char b));`
- (n) `int (*pf)(void);`
- (o) `int (*pf)(char a, char b);`
- (p) `int (*pf)(char *a, char *b);`

- 10.45** Write an appropriate declaration for each of the following situations.

- (a) Declare two pointers whose objects are the integer variables i and j.
- (b) Declare a pointer to a floating-point quantity, and a pointer to a double-precision quantity.
- (c) Declare a function that accepts two integer arguments and returns a pointer to a long integer.
- (d) Declare a function that accepts two arguments and returns a long integer. Each argument will be a pointer to an integer quantity.
- (e) Declare a one-dimensional floating-point array using pointer notation.
- (f) Declare a two-dimensional floating-point array, with 15 rows and 30 columns, using pointer notation.
- (g) Declare an array of strings whose initial values are “red,” “green” and “blue.”
- (h) Declare a function that accepts another function as an argument and returns a pointer to a character. The function passed as an argument will accept an integer argument and return an integer quantity.
- (i) Declare a pointer to a function that accepts three integer arguments and returns a floating-point quantity.
- (j) Declare a pointer to a function that accepts three pointers to integer quantities as arguments and returns a pointer to a floating-point quantity.

10.46 A C program contains the following statements.

```
char u, v = 'A';
char *pu, *pv = &v;

. . . .

*pv = v + 1;
u = *pv + 1;
pu = &u;
```

Suppose each character occupies 1 byte of memory. If the value assigned to *u* is stored in (hexadecimal) address F8C and the value assigned to *v* is stored in address F8D, then

- (a) What value is represented by *&v*?
- (b) What value is assigned to *pv*?
- (c) What value is represented by **pv*?
- (d) What value is assigned to *u*?
- (e) What value is represented by *&u*?
- (f) What value is assigned to *pu*?
- (g) What value is represented by **pu*?

10.47 A C program contains the following statements.

```
int i, j = 25;
int *pi, *pj = &j;

. . . .

*pj = j + 5;
i = *pj + 5;
pi = pj;
*pi = i + j;
```

Suppose each integer quantity occupies 2 bytes of memory. If the value assigned to *i* begins at (hexadecimal) address F9C and the value assigned to *j* begins at address F9E, then

- (a) What value is represented by *&i*?
- (b) What value is represented by *&j*?
- (c) What value is assigned to *pj*?
- (d) What value is assigned to **pj*?
- (e) What value is assigned to *i*?
- (f) What value is represented by *pi*?
- (g) What final value is assigned to **pi*?
- (h) What value is represented by *(pi + 2)*?
- (i) What value is represented by the expression *(*pi + 2)*?
- (j) What value is represented by the expression **(pi + 2)*?

10.48 A C program contains the following statements.

```
float a = 0.001, b = 0.003;
float c, *pa, *pb;

pa = &a;
*pa = 2 * a;
pb = &b;
c = 3 * (*pb - *pa);
```

Suppose each floating-point number occupies 4 bytes of memory. If the value assigned to *a* begins at (hexadecimal) address 1130, the value assigned to *b* begins at address 1134, and the value assigned to *c* begins at 1138, then

- (a) What value is assigned to *&a*?
- (b) What value is assigned to *&b*?
- (c) What value is assigned to *&c*?
- (d) What value is assigned to *pa*?
- (e) What value is represented by **pa*?
- (f) What value is represented by *&(*pa)*?
- (g) What value is assigned to *pb*?
- (h) What value is represented by **pb*?
- (i) What value is assigned to *c*?

10.49 The skeletal structure of a C program is shown below.

```

int funct1(char a, char b);
int funct2(char *pa, char *pb);

main()
{
    char a = 'X';
    char b = 'Y';
    int i, j;

    . . .

    i = funct1(a, b);
    printf("a=%c    b=%c\n", a, b);

    . . .

    j = funct2(&a, &b);
    printf("a=%c    b=%c", a, b);
}

int funct1(char c1, char c2)
{
    c1 = 'P';
    c2 = 'Q';

    . . .

    return((c1 < c2) ? c1 : c2);
}

int funct2(char *c1, char *c2)
{
    *c1 = 'P';
    *c2 = 'Q';

    . . .

    return((*c1 == *c2) ? *c1 : *c2);
}

```

- (a) Within main, what value is assigned to *i*?
- (b) What value is assigned to *j*?

- (c) What values are displayed by the first `printf` statement?
- (d) What values are displayed by the second `printf` statement?

Assume ASCII characters.

10.50 The skeletal structure of a C program is shown below.

```
void funct(int *p);

main()
{
    static int a[5] = {10, 20, 30, 40, 50};

    . . . .

    funct(a);

    . . . .

}

void funct(int *p)
{
    int i, sum = 0;
    for (i = 0; i < 5; ++i)
        sum += *(p + i);
    printf("sum=%d", sum);
    return;
}
```

- (a) What kind of argument is passed to `funct`?
- (b) What kind of information is returned by `funct`?
- (c) What kind of formal argument is defined within `funct`?
- (d) What is the purpose of the `for` loop that appears within `funct`?
- (e) What value is displayed by the `printf` statement within `funct`?

10.51 The skeletal structure of a C program is shown below.

```
void funct(int *p);

main()
{
    static int a[5] = {10, 20, 30, 40, 50};

    . . . .

    funct(a + 3);

    . . . .

}

void funct(int *p)
{
    int i, sum = 0;
    for (i = 3; i < 5; ++i)
        sum += *(p + i);
    printf("sum=%d", sum);
    return;
}
```

- (a) What kind of argument is passed to **funct**?
- (b) What kind of information is returned by **funct**?
- (c) What information is actually passed to **funct**?
- (d) What is the purpose of the **for** loop that appears within **funct**?
- (e) What value is displayed by the **printf** statement within **funct**?

Compare your answers with those of the previous problem. In what ways do these two skeletal outlines differ?

10.52 The skeletal structure of a C program is shown below.

```

int *funct(int *p);

main()
{
    static int a[5] = {10, 20, 30, 40, 50};
    int *ptmax;

    . . . .

    ptmax = funct(a);
    printf("max=%d", *ptmax);

    . . . .

}

int *funct(int *p)
{
    int i, imax, max = 0;
    for (i = 0; i < 5; ++i)
        if (*(p + i) > max) {
            max = *(p + i);
            imax = i;
        }
    return(p + imax);
}

```

- (a) Within **main**, what is **ptmax**?
- (b) What kind of information is returned by **funct**?
- (c) What is assigned to **ptmax** when the function is accessed?
- (d) What is the purpose of the **for** loop that appears within **funct**?
- (e) What value is displayed by the **printf** statement within **main**?

Compare your answers with those of the previous two problems. In what ways are the skeletal outlines different?

10.53 A C program contains the following declaration.

```
static int x[8] = {10, 20, 30, 40, 50, 60, 70, 80};
```

- (a) What is the meaning of **x**?
- (b) What is the meaning of **(x + 2)**?
- (c) What is the value of ***x**?
- (d) What is the value of **(*x + 2)**?
- (e) What is the value of ***(x + 2)**?

10.54 A C program contains the following declaration.

```
static float table[2][3] = {
    {1.1, 1.2, 1.3},
    {2.1, 2.2, 2.3}
};
```

(a) What is the meaning of `table`?
 (b) What is the meaning of `(table + 1)`?
 (c) What is the meaning of `*(table + 1)`?
 (d) What is the meaning of `(*(table + 1) + 1)`?
 (e) What is the meaning of `(*(table) + 1)`?
 (f) What is the value of `*(*(table + 1) + 1)`?
 (g) What is the value of `*(*(table) + 1)`?
 (h) What is the value of `*(*(*(table + 1)))`?
 (i) What is the value of `*(*(*(table) + 1) + 1)`?

10.55 A C program contains the following declaration.

```
static char *color[6] = {"red", "green", "blue", "white", "black", "yellow"};
```

(a) What is the meaning of `color`?
 (b) What is the meaning of `(color + 2)`?
 (c) What is the value of `*color`?
 (d) What is the value of `*(color + 2)`?
 (e) How do `color[5]` and `*(color + 5)` differ?

10.56 The skeletal structure of a C program is shown below.

```
float one(float x, float y);
float two(float x, float y);
float three(float (*pt)(float x, float y));

main()
{
    float a, b;
    . . .
    a = three(one);
    . . .
    b = three(two);
    . . .
}

float one(float x, float y)
{
    float z;
    z = . . .
    return(z);
}
```

```

float two(float p, float q)
{
    float r;
    r = . . . . .
    return(r);
}

float three(float (*pt)(float x, float y))
{
    float a, b, c;
    . . . .
    c = (*pt)(a, b);
    . . . .
    return(c);
}

```

- (a) Interpret each of the function prototypes.
- (b) Interpret the definitions of the functions `one` and `two`.
- (c) Interpret the definition of the function `three`. How does `three` differ from `one` and `two`?
- (d) What happens within `main` each time `three` is accessed?

10.57 The skeletal structure of a C program is shown below.

```

float one(float *px, float *py);
float two(float *px, float *py);
float *three(float (*pt)(float *px, float *py));

main()
{
    float *pa, *pb;
    . . . .
    pa = three(one);
    . . . .
    pb = three(two);
    . . . .
}

float one(float *px, float *py)
{
    float z;
    z = . . . .
    return(z);
}

```

```

float two(float *pp, float *pq)
{
    float r;
    r = . . . . .
    return(r);
}

float *three(float (*pt)(float *px, float *py))
{
    float a, b, c;
    . . . .
    c = (*pt)(&a, &b);
    . . . .
    return(&c);
}

```

- (a) Interpret each of the function prototypes.
- (b) Interpret the definitions of the functions one and two.
- (c) Interpret the definition of the function three. How does three differ from one and two?
- (d) What happens within main each time three is accessed?
- (e) How does this program outline differ from the outline shown in the last example?

10.58 Explain the purpose of each of the following declarations.

- (a) float (*x)(int *a);
- (b) float (*x(int *a))[20];
- (c) float x(int (*a)[]);
- (d) float x(int *a[]);
- (e) float *x(int a[]);
- (f) float *x(int (*a)[]);
- (g) float *x(int *a[]);
- (h) float (*x)(int (*a)[]);
- (i) float *(*x)(int *a[]);
- (j) float (*x[20])(int a);
- (k) float *(*x[20])(int *a);

10.59 Write an appropriate declaration for each of the following situations involving pointers.

- (a) Declare a function that accepts an argument which is a pointer to an integer quantity and returns a pointer to a six-element character array.
- (b) Declare a function that accepts an argument which is a pointer to an integer array and returns a character.
- (c) Declare a function that accepts an argument which is an array of pointers to integer quantities and returns a character.
- (d) Declare a function that accepts an argument which is an integer array and returns a pointer to a character.
- (e) Declare a function that accepts an argument which is a pointer to an integer array and returns a pointer to a character.
- (f) Declare a function that accepts an argument which is an array of pointers to integer quantities and returns a pointer to a character.

- (g) Declare a pointer to a function that accepts an argument which is a pointer to an integer array and returns a character.
- (h) Declare a pointer to a function that accepts an argument which is a pointer to an integer array and returns a pointer to a character.
- (i) Declare a pointer to a function that accepts an argument which is an array of pointers to integer quantities and returns a pointer to a character.
- (j) Declare a 12-element array of pointers to functions. Each function will accept two double-precision quantities as arguments and will return a double-precision quantity.
- (k) Declare a 12-element array of pointers to functions. Each function will accept two double-precision quantities as arguments and will return a pointer to a double-precision quantity.
- (l) Declare a 12-element array of pointers to functions. Each function will accept two pointers to double-precision quantities as arguments and will return a pointer to a double-precision quantity.

Programming Problems

10.60 Modify the program shown in Example 10.1 as follows.

- (a) Use floating-point data rather than integer data. Assign an initial value of 0.3 to `u`.
- (b) Use double-precision data rather than integer data. Assign an initial value of 0.3×10^{45} to `u`.
- (c) Use character data rather than integer data. Assign an initial value of 'C' to `u`.

Execute each modification and compare the results with those given in Example 10.1. Be sure to modify the `printf` statements accordingly.

10.61 Modify the program shown in Example 10.3 as follows.

- (a) Use floating-point data rather than integer data. Assign an initial value of 0.3 to `v`.
- (b) Use double-precision data rather than integer data. Assign an initial value of 0.3×10^{45} to `v`.
- (c) Use character data rather than integer data. Assign an initial value of 'C' to `v`.

Execute each modification and compare the results with those given in Example 10.3. Be sure to modify the `printf` statements accordingly.

10.62 Modify the program shown in Example 10.7 so that a single one-dimensional, character-type array is passed to `funct1`. Delete `funct2` and all references to `funct2`. Initially, assign the string "red" to the array within `main`. Then reassign the string "green" to the array within `funct1`. Execute the program and compare the results with those shown in Example 10.7. Remember to modify the `printf` statements accordingly.

10.63 Modify the program shown in Example 10.8 (analyzing a line of text) so that it also counts the number of words and the total number of characters in the line of text. (*Note:* A new word can be recognized by the presence of a blank space followed by a nonwhitespace character.) Test the program using the text given in Example 10.8.

10.64 Modify the program shown in Example 10.8 (analyzing a line of text) so that it can process multiple lines of text. First enter and store all lines of text. Then determine the number of vowels, consonants, digits, whitespace characters and "other" characters for each line. Finally, determine the average number of vowels per line, consonants per line, etc. Write and execute the program two different ways.

- (a) Store the multiple lines of text in a two-dimensional array of characters.
- (b) Store the multiple lines of text as individual strings whose maximum length is unspecified. Maintain a pointer to each string within a one-dimensional array of pointers.

In each case, identify the last line of text in some predetermined manner (e.g., by entering the string "END"). Test the program using several lines of text of your own choosing.

10.65 Modify the program shown in Example 10.12 so that the elements of `x` are long integers rather than ordinary integers. Execute the program and compare the results with those shown in Example 10.12. (Remember to modify the `printf` statement to accommodate the long integer quantities.)

10.66 Modify the program shown in Example 10.16 so that any one of the following rearrangements can be carried out:

- (a) Smallest to largest, by magnitude
- (b) Smallest to largest, algebraic
- (c) Largest to smallest, by magnitude
- (d) Largest to smallest, algebraic

Use pointer notation to represent individual integer quantities, as in Example 10.16. (Recall that an array version of this problem was presented in Example 9.13.) Include a menu that will allow the user to select which rearrangement will be used each time the program is executed. Test the program using the following 10 values.

| | |
|------|-------|
| 4.7 | -8.0 |
| -2.3 | 11.4 |
| 12.9 | 5.1 |
| 8.8 | -0.2 |
| 6.0 | -14.7 |

10.67 Modify the program shown in Example 10.22 (adding two tables of numbers) so that each element in the table **c** is the larger of the corresponding elements in tables **a** and **b** (rather than the sum of the corresponding elements in **a** and **b**). Represent each table (each array) as a pointer to a group of one-dimensional arrays, as in Example 10.22. Use pointer notation to access the individual table elements. Test the program using the tabular data provided in Example 9.19. (You may wish to experiment with this program, using several different ways to represent the arrays and the individual array elements.)

10.68 Repeat the previous problem, representing each table (each array) as a one-dimensional array of pointers, as discussed in Example 10.24.

10.69 Modify the program shown in Example 10.26 (reordering a list of strings) so that the list of strings can be rearranged into either alphabetical or reverse-alphabetical order. Use pointer notation to represent the beginning of each string. Include a menu that will allow the user to select which rearrangement will be used each time the program is executed. Test the program using the data provided in Example 9.20.

10.70 Modify the program shown in Example 10.28 (displaying the day of the year) so that it can determine the number of days between two dates, assuming both dates are beyond the base date of January 1, 1900. (*Hint:* Determine the number of days between the first specified date and the base date; then determine the number of days between the second specified date and the base date. Finally, determine the difference between these two calculated values.)

10.71 Modify the program shown in Example 10.30 (compound interest calculations) so that it generates a table of F-values for various interest rates, using different compounding frequencies. Assume that A and n are input values. Display the output in the following manner.

| |
|---|
| A = . . . |
| n = . . . |
| Interest rate = 5% 6% 7% 8% 9% 10% 11% 12% 13% 14% 15% |
| Frequency of Compounding |
| Annual — — — — — — — — — — — |
| Semiannual — — — — — — — — — — — |
| Quarterly — — — — — — — — — — — |
| Monthly — — — — — — — — — — — |
| Daily — — — — — — — — — — — |
| Continuously — — — — — — — — — — — |

Notice that the first four rows are generated by one function with different arguments, and each of the last two rows is generated by a different function.

- 10.72** Modify the program shown in Example 10.30 (compound interest calculations) so that it generates a table of F-values for various time periods, using different compounding frequencies. Assume that A and i are input values. Display the output in the following manner.

| | | | | | | | | | | |
|--------------------------|-----------|---|---|---|---|---|---|---|---|----|
| | A = . . . | | | | | | | | | |
| | i = . . . | | | | | | | | | |
| Time period (n) = | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| Frequency of Compounding | | | | | | | | | | |
| Annual | — | — | — | — | — | — | — | — | — | — |
| Semiannual | — | — | — | — | — | — | — | — | — | — |
| Quarterly | — | — | — | — | — | — | — | — | — | — |
| Monthly | — | — | — | — | — | — | — | — | — | — |
| Daily | — | — | — | — | — | — | — | — | — | — |
| Continuously | — | — | — | — | — | — | — | — | — | — |

Notice that the first four rows are generated by one function with different arguments, and each of the last two rows is generated by a different function.

- 10.73** Repeat the previous problem, but transpose the table so that each row represents a different value for n and each column represents a different compounding frequency. Consider integer values of n ranging from 1 to 50. Note that this table will consist of 50 rows and 6 columns. (*Hint:* Generate the table by columns, storing each column in a two-dimensional array. Display the entire array after all the values have been generated.)

Compare the programming effort required for this problem with the programming effort required for the preceding problem.

- 10.74** Examples 9.8 and 9.9 present programs to calculate the average of a list of numbers and then calculate the deviations about the average. Both programs make use of one-dimensional, floating-point arrays. Modify both programs so that they utilize pointer notation. (Note that the program shown in Example 9.9 includes the assignment of initial values to individual array elements.) Test both programs using the data given in the examples.

- 10.75** Modify the program given in Example 9.14 (piglatin generator) so that it uses character-type arrays. Modify the program so that it uses pointer notation. Test the program using several lines of text of your own choosing.

- 10.76** Write a complete C program, using pointer notation in place of arrays, for each of the following problems taken from the end of Chap. 9.

- (a) Problem 9.39 (read a line of text, store it within the computer's memory, and then display it backwards).
- (b) Problem 9.40 (process a set of student exam scores). Test the program using the data given in Prob. 9.40.
- (c) Problem 9.42 (process a set of weighted student exam scores, and calculate the deviation of each student's average about the overall class average). Test the program using the data given in Prob. 9.40.
- (d) Problem 9.44 (generate a table of compound interest factors).
- (e) Problem 9.45 (convert from one foreign currency to another).
- (f) Problem 9.46 (determine the capital for a specified country, or the country whose capital is specified). Test the program using the list of countries and their capitals given in Prob. 9.46.
- (g) Problem 9.47(a) (matrix/vector multiplication). Test the program using the data given in Prob. 9.47(a).
- (h) Problem 9.47(b) (matrix multiplication). Test the program using the data given in Prob. 9.47(b).
- (i) Problem 9.47(d) (Lagrange interpolation). Test the program using the data given in Prob. 9.47(d).
- (j) Problem 9.48(a) (blackjack).

- (k) Problem 9.48(b) (roulette).
- (l) Problem 9.48(c) (BINGO).
- (m) Problem 9.49 (encode and decode a line of text).

10.77 Write a complete C program, using pointer notation, that will generate a table containing the following three columns:

$$t \quad ae^{bt} \sin ct \quad ae^{bt} \cos ct$$

Structure the program in the following manner: write two special functions, `f1` and `f2`, where `f1` evaluates the quantity $ae^{bt} \sin ct$ and `f2` evaluates $ae^{bt} \cos ct$. Have `main` enter the values of a , b and c , and then call a function, `table_gen`, which will generate the actual table. Pass `f1` and `f2` to `table_gen` as arguments.

Test the program using the values $a = 2$, $b = -0.1$, $c = 0.5$ where the values of t are $1, 2, 3, \dots, 60$.

Chapter 11

Structures and Unions

In Chap. 9 we studied the array, which is a data structure whose elements are all of the same data type. We now turn our attention to the *structure*, in which the individual elements can differ in type. Thus, a single structure might contain integer elements, floating-point elements and character elements. Pointers, arrays and other structures can also be included as elements within a structure. The individual structure elements are referred to as *members*.

This chapter is concerned with the use of structures within a C program. We will see how structures are defined, and how their individual members are accessed and processed within a program. The relationships between structures and pointers, arrays and functions will also be examined.

Closely associated with the structure is the *union*, which also contains multiple members. Unlike a structure, however, the members of a union share the same storage area, even though the individual members may differ in type. Thus, a union permits several different data items to be stored in the same portion of the computer's memory at different times. We will see how unions are defined and utilized within a C program.

11.1 DEFINING A STRUCTURE

Structure declarations are somewhat more complicated than array declarations, since a structure must be defined in terms of its individual members. In general terms, the composition of a structure may be defined as

```
struct tag {  
    member 1;  
    member 2;  
    . . . . .  
    member m;  
};
```

In this declaration, **struct** is a required keyword; *tag* is a name that identifies structures of this type (i.e., structures having this composition); and *member 1*, *member 2*, . . . , *member m* are individual member declarations. (Note: There is no formal distinction between a structure *definition* and a structure *declaration*; the terms are used interchangeably.)

The individual members can be ordinary variables, pointers, arrays, or other structures. The member names within a particular structure must be distinct from one another, though a member name can be the same as the name of a variable that is defined outside of the structure. A storage class, however, cannot be assigned to an individual member, and individual members cannot be initialized within a structure type declaration.

Once the composition of the structure has been defined, individual structure-type variables can be declared as follows:

```
storage-class struct tag variable 1, variable 2, . . . , variable n;
```

where *storage-class* is an optional storage class specifier, **struct** is a required keyword, *tag* is the name that appeared in the structure declaration, and *variable 1*, *variable 2*, . . . , *variable n* are structure variables of type *tag*.

EXAMPLE 11.1 A typical structure declaration is shown below.

```
struct account {
    int acct_no;
    char acct_type;
    char name[80];
    float balance;
};
```

This structure is named **account** (i.e., the tag is **account**). It contains four members: an integer quantity (**acct_no**), a single character (**acct_type**), an 80-element character array (**name[80]**), and a floating-point quantity (**balance**). The composition of this account is illustrated schematically in Fig. 11.1.

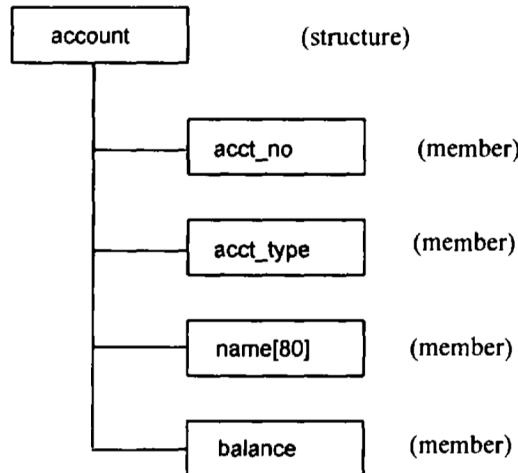


Fig. 11.1

We can now declare the structure variables **oldcustomer** and **newcustomer** as follows.

```
struct account oldcustomer, newcustomer;
```

Thus, **oldcustomer** and **newcustomer** are variables of type **account**. In other words, **oldcustomer** and **newcustomer** are structure-type variables whose composition is identified by the tag **account**.

It is possible to combine the declaration of the structure composition with that of the structure variables, as shown below.

```
storage-class struct tag {
    member 1;
    member 2;
    . . .
    member m;
} variable 1, variable 2, . . . , variable n;
```

The **tag** is optional in this situation.

EXAMPLE 11.2 The following single declaration is equivalent to the two declarations presented in the previous example.

```

struct account {
    int acct_no;
    char acct_type;
    char name[80];
    float balance;
} oldcustomer, newcustomer;

```

Thus, `oldcustomer` and `newcustomer` are structure variables of type `account`.

Since the variable declarations are now combined with the declaration of the structure type, the tag (i.e., `account`) need not be included. Thus, the above declaration can also be written as

```

struct {
    int acct_no;
    char acct_type;
    char name[80];
    float balance;
} oldcustomer, newcustomer;

```

A structure variable may be defined as a member of another structure. In such situations, the declaration of the embedded structure must appear before the declaration of the outer structure.

EXAMPLE 11.3 A C program contains the following structure declarations.

```

struct date {
    int month;
    int day;
    int year;
};

struct account {
    int acct_no;
    char acct_type;
    char name[80];
    float balance;
    struct date lastpayment;
} oldcustomer, newcustomer;

```

The second structure (`account`) now contains another structure (`date`) as one of its members. Note that the declaration of `date` precedes the declaration of `account`. The composition of `account` is shown schematically in Fig. 11.2.

The members of a structure variable can be assigned initial values in much the same manner as the elements of an array. The initial values must appear in the order in which they will be assigned to their corresponding structure members, enclosed in braces and separated by commas. The general form is

storage-class `struct tag variable = {value 1, value 2, . . . , value m};`

where `value 1` refers to the value of the first member, `value 2` refers to the value of the second member, and so on. A structure variable, like an array, can be initialized only if its storage class is either `external` or `static`.

EXAMPLE 11.4 This example illustrates the assignment of initial values to the members of a structure variable.

```

struct date {
    int month;
    int day;
    int year;
};

```

```

struct account {
    int acct_no;
    char acct_type;
    char name[80];
    float balance;
    struct date lastpayment;
};

static struct account customer = {12345, 'R', "John W. Smith", 586.30, 5, 24, 90};

```

Thus, `customer` is a static structure variable of type `account`, whose members are assigned initial values. The first member (`acct_no`) is assigned the integer value 12345, the second member (`acct_type`) is assigned the character 'R', the third member (`name[80]`) is assigned the string "John W. Smith", and the fourth member (`balance`) is assigned the floating-point value 586.30. The last member is itself a structure that contains three integer members (`month`, `day` and `year`). Therefore, the last member of `customer` is assigned the integer values 5, 24 and 90.

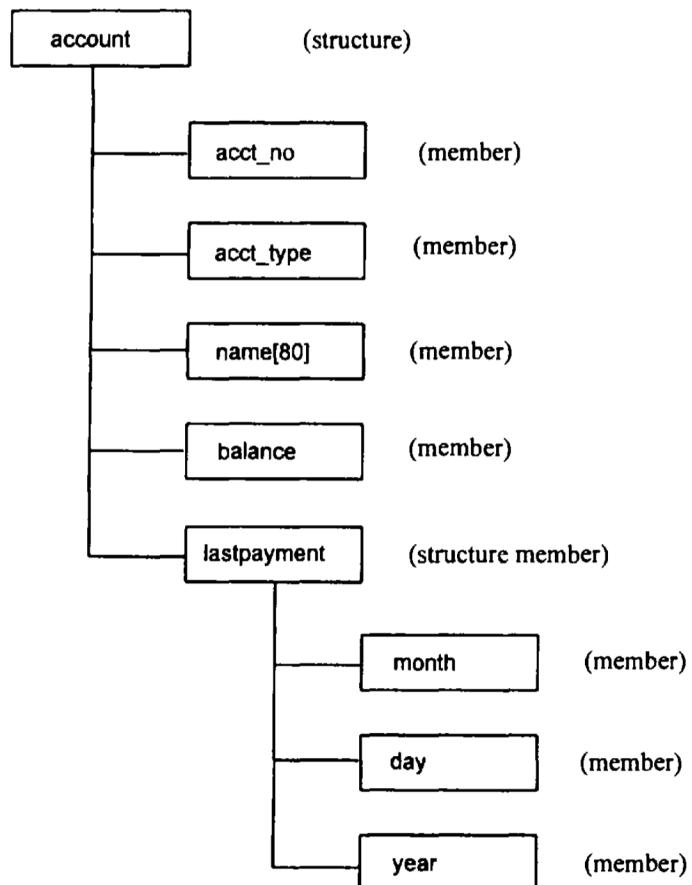


Fig. 11.2

It is also possible to define an array of structures; i.e., an array in which each element is a structure. The procedure is illustrated in the following example.

EXAMPLE 11.5 A C program contains the following structure declarations.

```
struct date {
    int month;
    int day;
    int year;
};

struct account {
    int acct_no;
    char acct_type;
    char name[80];
    float balance;
    struct date lastpayment;
} customer[100];
```

In this declaration `customer` is a 100-element array of structures. Hence, each element of `customer` is a separate structure of type `account` (i.e., each element of `customer` represents an individual customer record).

Note that each structure of type `account` includes an array (`name[80]`) and another structure (`date`) as members. Thus, we have an array and a structure embedded within another structure, which is itself an element of an array.

It is, of course, also permissible to define `customer` in a separate declaration, as shown below.

```
struct date {
    int month;
    int day;
    int year;
};

struct account {
    int acct_no;
    char acct_type;
    char name[80];
    float balance;
    struct date lastpayment;
};

struct account customer[100];
```

An array of structures can be assigned initial values just as any other array. Remember that each array element is a structure that must be assigned a corresponding set of initial values, as illustrated below.

EXAMPLE 11.6 A C program contains the following declarations.

```
struct date {
    char name[80];
    int month;
    int day;
    int year;
};

static struct date birthday[] = {"Amy", 12, 30, 73,
                                 "Gail", 5, 13, 66,
                                 "Marc", 7, 15, 72,
                                 "Marla", 11, 29, 70,
                                 "Megan", 2, 4, 77,
                                 "Sharon", 12, 29, 63,
                                 "Susan", 4, 12, 69};
```

In this example `birthday` is an array of structures whose size is unspecified. The initial values will define the size of the array, and the amount of memory required to store the array.

Notice that each row in the variable declaration contains four constants. These constants represent the initial values, i.e., the name, month, day and year, for one array element. Since there are 7 rows (7 sets of constants), the array will contain 7 elements, numbered 0 to 6.

Some programmers may prefer to embed each set of constants within a separate pair of braces, in order to delineate the individual array elements more clearly. This is entirely permissible. Thus, the array declaration can be written

```
static struct date b[.thday[]] = {
    {"Amy", 12, 30, 73},
    {"Gail", 5, 13, 66},
    {"Marc", 7, 15, 72},
    {"Marla", 11, 29, 70},
    {"Megan", 2, 4, 77},
    {"Sharon", 12, 29, 63},
    {"Susan", 4, 12, 69}
};
```

Remember that each structure is a self-contained entity with respect to member definitions. Thus, the same member name can be used in different structures to represent different data. In other words, the scope of a member name is confined to the particular structure within which it is defined.

EXAMPLE 11.7 Two different structures, called `first` and `second`, are declared below.

```
struct first {
    float a;
    int b;
    char c;
};

struct second {
    char a;
    float b, c;
};
```

Notice that the individual member names `a`, `b` and `c` appear in both structure declarations, but the associated data types are different. Thus, `a` represents a floating-point quantity in `first` and a character in `second`. Similarly, `b` represents an integer quantity in `first` and a floating-point quantity in `second`, whereas `c` represents a character in `first` and a floating-point quantity in `second`. This duplication of member names is permissible, since the scope of each set of member definitions is confined to its respective structure. Within each structure the member names are distinct, as required.

11.2 PROCESSING A STRUCTURE

The members of a structure are usually processed individually, as separate entities. Therefore, we must be able to access the individual structure members. A structure member can be accessed by writing

variable.member

where *variable* refers to the name of a structure-type variable, and *member* refers to the name of a member within the structure. Notice the period (.) that separates the variable name from the member name. This period is an operator; it is a member of the highest precedence group, and its associativity is left to right (see Appendix C).

EXAMPLE 11.8 Consider the following structure declarations.

```
struct date {
    int month;
    int day;
    int year;
};

struct account {
    int acct_no;
    char acct_type;
    char name[80];
    float balance;
    struct date lastpayment;
} customer;
```

In this example `customer` is a structure variable of type `account`. If we wanted to access the customer's account number, we would write

```
customer.acct_no
```

Similarly, the customer's name and the customer's balance can be accessed by writing

```
customer.name
```

and

```
customer.balance
```

Since the period operator is a member of the highest precedence group, this operator will take precedence over the unary operators as well as the various arithmetic, relational, logical and assignment operators. Thus, an expression of the form `++variable.member` is equivalent to `++(variable.member)`; i.e., the `++` operator will apply to the structure member, not the entire structure variable. Similarly, the expression `&variable.member` is equivalent to `&(variable.member)`; thus, the expression accesses the address of the structure member, not the starting address of the structure variable.

EXAMPLE 11.9 Consider the structure declarations given in Example 11.8; i.e.,

```
struct date {
    int month;
    int day;
    int year;
};

struct account {
    int acct_no;
    char acct_type;
    char name[80];
    float balance;
    struct date lastpayment;
} customer;
```

Several expressions involving the structure variable `customer` and its members are given below.

| <u>Expression</u> | <u>Interpretation</u> |
|------------------------------------|--|
| <code>++customer.balance</code> | Increment the value of <code>customer.balance</code> |
| <code>customer.balance++</code> | Increment the value of <code>customer.balance</code> after accessing its value |
| <code>--customer.acct_no</code> | Decrement the value of <code>customer.acct_no</code> |
| <code>&customer</code> | Access the beginning address of <code>customer</code> |
| <code>&customer.acct_no</code> | Access the address of <code>customer.acctno</code> |

More complex expressions involving the repeated use of the period operator may also be written. For example, if a structure member is itself a structure, then a member of the embedded structure can be accessed by writing

`variable.member.submember`

where `member` refers to the name of the member within the outer structure, and `submember` refers to the name of the member within the embedded structure. Similarly, if a structure member is an array, then an individual array element can be accessed by writing

`variable.member[expression]`

where `expression` is a nonnegative value that indicates the array element.

EXAMPLE 11.10 Consider once again the structure declarations presented in Example 11.8.

```
struct date {
    int month;
    int day;
    int year;
};

struct account {
    int acct_no;
    char acct_type;
    char name[80];
    float balance;
    struct date lastpayment;
} customer;
```

The last member of `customer` is `customer.lastpayment`, which is itself a structure of type `date`. To access the month of the last payment, we would therefore write

`customer.lastpayment.month`

Moreover, this value can be incremented by writing

`++customer.lastpayment.month`

Similarly, the third member of `customer` is the character array `customer.name`. The third character within this array can be accessed by writing

`customer.name[2]`

This character's address can be obtained as

`&customer.name[2]`

The use of the period operator can be extended to arrays of structures, by writing

array[expression].member

where *array* refers to the array name, and *array[expression]* is an individual array element (a structure variable). Therefore *array[expression].member* will refer to a specific member within a particular structure.

EXAMPLE 11.11 Consider the following structure declarations, which were originally presented in Example 11.5.

```
struct date {
    int month;
    int day;
    int year;
};

struct account {
    int acct_no;
    char acct_type;
    char name[80];
    float balance;
    struct date lastpayment;
} customer[100];
```

In this example *customer* is an array that may contain as many as 100 elements. Each element is a structure of type *account*. Thus, if we wanted to access the account number for the 14th customer (i.e., *customer[13]*, since the subscripts begin with 0), we would write *customer[13].acct_no*. Similarly, this customer's balance can be accessed by writing *customer[13].balance*. The corresponding address can be obtained as *&customer[13].balance*.

The 14th customer's name can be accessed by writing *customer[13].name*. Moreover, we can access an individual character within the name by specifying a subscript. For example, the 8th character within the customer's name can be accessed by writing *customer[13].name[7]*. In a similar manner we can access the month, day and year of the 14th customer's last payment by specifying the individual members of *customer[13].lastpayment*, i.e., *customer[13].lastpayment.month*, *customer[13].lastpayment.day*, *customer[13].lastpayment.year*. Moreover, the expression *++customer[13].lastpayment.day* causes the value of the day to be incremented.

Structure members can be processed in the same manner as ordinary variables of the same data type. Single-valued structure members can appear in expressions, they can be passed to functions, and they can be returned from functions, as though they were ordinary single-valued variables. Complex structure members are processed in the same way as ordinary data items of that same type. For example, a structure member that is an array can be processed in the same manner as an ordinary array, and with the same restrictions. Similarly, a structure member that is itself a structure can be processed on a member-by-member basis (the members here refer to the embedded structure), the same as any other structure.

EXAMPLE 11.12 Several statements or groups of statements that access individual structure members are shown below. All of the structure members conform to the declarations given in Example 11.8.

```
customer.balance = 0;

customer.balance -= payment;

customer.lastpayment.month = 12;

printf("Name: %s\n", customer.name);

if (customer.acct_type == 'P')
    printf("Preferred account no.: %d\n", customer.acct_no);
else
    printf("Regular account no.: %d\n", customer.acct_no);
```

The first statement assigns a value of zero to `customer.balance`, whereas the second statement causes the value of `customer.balance` to be decreased by the value of `payment`. The third statement causes the value 12 to be assigned to `customer.lastpayment.month`. Note that `customer.lastpayment.month` is a member of the embedded structure `customer.lastpayment`.

The fourth statement passes the array `customer.name` to the `printf` function, causing the customer name to be displayed. Finally, the last example illustrates the use of structure members in an `if - else` statement. Also, we see a situation in which the structure member `customer.acct_no` is passed to a function as an argument.

In some older versions of C, structures must be processed on a member-by-member basis. With this restriction, the only permissible operation on an entire structure is to take its address (more about this later). However, the current ANSI standard permits entire structures to be assigned to one another provided the structures have the same composition.

EXAMPLE 11.13 Suppose `oldcustomer` and `newcustomer` are structure variables having the same composition; i.e.,

```
struct date {
    int month;
    int day;
    int year;
};

struct account {
    int acct_no;
    char acct_type;
    char name[80];
    float balance;
    struct date lastpayment;
} oldcustomer, newcustomer;
```

as declared in Example 11.8. Let us assume that all of the members of `oldcustomer` have been assigned individual values. In most newer versions of C, it is possible to copy these values to `newcustomer` simply by writing

```
newcustomer = oldcustomer;
```

On the other hand, some older versions of C may require that the values be copied individually, member by member; for example,

```
newcustomer.acct_no = oldcustomer.acct_no;
newcustomer.acct_type = oldcustomer.acct_type;
. . .
newcustomer.lastpayment.year = oldcustomer.lastpayment.year;
```

It is also possible to pass entire structures to and from functions, though the way this is done varies from one version of C to another. Older versions of C allow only pointers to be passed, whereas the ANSI standard allows passing of the structures themselves. We will discuss this further in Sec. 11.5. Before moving on to the relationship between structures and pointers and the methods for passing structures to functions, however, let us consider a more comprehensive example that involves the processing of structure members.

EXAMPLE 11.14 Updating Customer Records To illustrate further how the individual members of a structure can be processed, consider a very simple customer billing system. In this system the customer records will be stored within an array of structures. Each record will be stored as an individual structure (i.e., as an array element) containing the customer's name, street address, city and state, account number, account status (current, overdue or delinquent), previous balance, current payment, new balance and payment date.

The overall strategy will be to enter each customer record into the computer, updating it as soon as it is entered, to reflect current payments. All of the updated records will then be displayed, along with the current status of each account. The account status will be based upon the size of the current payment relative to the customer's previous balance.

The structure declarations are shown below.

```
struct date {
    int month;
    int day;
    int year;
};

struct account {
    char name[80];
    char street[80];
    char city[80];
    int acct_no;
    char acct_type;
    float oldbalance;
    float newbalance;
    float payment;
    struct date lastpayment;
} customer[100];
```

Notice that **customer** is a 100-element array of structures. Thus, each array element (each structure) will represent one customer record. Each structure includes three members that are character-type arrays (**name**, **street** and **city**), and one member that is another structure (**lastpayment**).

The status of each account will be determined in the following manner:

1. If the current payment is greater than zero but less than 10 percent of the previous outstanding balance, the account will be overdue.
2. If there is an outstanding balance and the current payment is zero, the account will be delinquent.
3. Otherwise, the account will be current.

The overall program strategy will be as follows.

1. Specify the number of customer accounts (i.e., the number of structures) to be processed.
2. For each customer, read in the following items.

| | |
|--|---|
| (a) name (b) street (c) city (d) account number | (e) previous balance (f) current payment (g) payment date |
|--|---|
3. As each customer record is read into the computer, update it in the following manner.
 - (a) Compare the current payment with the previous balance and determine the appropriate account status.
 - (b) Calculate a new account balance by subtracting the current payment from the previous balance (a negative balance will indicate a credit).
4. After all of the customer records have been entered and processed, write out the following information for each customer.

| | |
|--|---|
| (a) name (b) account number (c) street (d) city | (e) old balance (f) current payment (g) new balance (h) account status |
|--|---|

Let us write the program in a modular manner, with one function to enter and update each record and another function to display the updated data. Ideally, we would like to pass every customer record (i.e., every array element) to each of these functions. Since each customer record is a structure, however, and we have not yet discussed how to pass a

structure to or from a function, we will define the array of structures as an external array. This will allow us to access the array elements, and the individual structure members, directly from all of the functions.

The individual program modules are straightforward, though some care is required in reading the individual structure members into the computer. Here is the entire program.

```
/* update a series of customer accounts (simplified billing system) */
/* maintain the customer accounts as an external array of structures */

#include <stdio.h>

void readinput(int i);
void writeoutput(int i);

struct date {
    int month;
    int day;
    int year;
};

struct account {
    char name[80];
    char street[80];
    char city[80];
    int acct_no;           /* (positive integer) */
    char acct_type;        /* C (current), O (overdue), or D (delinquent) */
    float oldbalance;      /* (nonnegative quantity) */
    float newbalance;      /* (nonnegative quantity) */
    float payment;         /* (nonnegative quantity) */
    struct date lastpayment;
} customer[100];          /* maintain as many as 100 customers */

main()
{
    int i, n;

    printf("CUSTOMER BILLING SYSTEM\n\n");
    printf("How many customers are there? ");
    scanf("%d", &n);

    for (i = 0; i < n; ++i) {
        readinput(i);

        /* determine account status */

        if (customer[i].payment > 0)
            customer[i].acct_type =
                (customer[i].payment < 0.1 * customer[i].oldbalance) ? 'O' : 'C';
        else
            customer[i].acct_type =
                (customer[i].oldbalance > 0) ? 'D' : 'C';

        /* adjust account balance */

        customer[i].newbalance = customer[i].oldbalance - customer[i].payment;
    };

    for (i = 0; i < n; ++i)
        writeoutput(i);
}
```

```
void readinput(int i)

/* read input data and update record for each customer */

{
    printf("\nCustomer no. %d\n", i + 1);
    printf("    Name: ");
    scanf(" %[^\n]", customer[i].name);
    printf("    Street: ");
    scanf(" %[^\n]", customer[i].street);
    printf("    City: ");
    scanf(" %[^\n]", customer[i].city);
    printf("    Account number: ");
    scanf("%d", &customer[i].acct_no);
    printf("    Previous balance: ");
    scanf("%f", &customer[i].oldbalance);
    printf("    Current payment: ");
    scanf("%f", &customer[i].payment);
    printf("    Payment date (mm/dd/yyyy): ");
    scanf("%d/%d/%d", &customer[i].lastpayment.month,
          &customer[i].lastpayment.day,
          &customer[i].lastpayment.year);
    return;
}

void writeoutput(int i)

/* display current information for each customer */

{
    printf("\nName:  %s", customer[i].name);
    printf("    Account number: %d\n", customer[i].acct_no);
    printf("Street: %s\n", customer[i].street);
    printf("City:  %s\n\n", customer[i].city);
    printf("Old balance: %7.2f", customer[i].oldbalance);
    printf("    Current payment: %7.2f", customer[i].payment);
    printf("    New balance: %7.2f\n\n", customer[i].newbalance);
    printf("Account status: ");

    switch (customer[i].acct_type)  {
    case 'C':
        printf("CURRENT\n\n");
        break;
    case 'O':
        printf("OVERDUE\n\n");
        break;
    case 'D':
        printf("DELINQUENT\n\n");
        break;
    default:
        printf("ERROR\n\n");
    }
    return;
}
```

Now suppose the program is used to process four fictitious customer records. The input dialog is shown below, with the user's responses underlined.

CUSTOMER BILLING SYSTEM

How many customers are there? 4

Customer no. 1

Name: Steve Johnson
Street: 123 Mountainview Drive
City: Denver, CO
Account number: 4208
Previous balance: 247.88
Current payment: 25.00
Payment date (mm/dd/yyyy): 6/14/1998

Customer no. 2

Name: Susan Richards
Street: 4383 Alligator Blvd
City: Fort Lauderdale, FL
Account number: 2219
Previous balance: 135.00
Current payment: 135.00
Payment date (mm/dd/yyyy): 8/10/2000

Customer no. 3

Name: Martin Peterson
Street: 1787 Pacific Parkway
City: San Diego, CA
Account number: 8452
Previous balance: 387.42
Current payment: 35.00
Payment date (mm/dd/yyyy): 9/22/1999

Customer no. 4

Name: Phyllis Smith
Street: 1000 Great White Way
City: New York, NY
Account number: 711
Previous balance: 260.00
Current payment: 0
Payment date (mm/dd/yyyy): 11/27/2001

The program will then generate the following output data:

Name: Steve Johnson Account number: 4208
Street: 123 Mountainview Drive
City: Denver, CO

Old balance: 247.88 Current payment: 25.00 New balance: 222.88

Account status: CURRENT

Name: Susan Richards Account number: 2219
Street: 4383 Alligator Blvd
City: Fort Lauderdale, FL

Old balance: 135.00 Current payment: 135.00 New balance: 0.00

Account status: CURRENT

Name: Martin Peterson Account number: 8452
Street: 1787 Pacific Parkway
City: San Diego, CA

Old balance: 387.42 Current payment: 35.00 New balance: 352.42

Account status: OVERDUE

Name: Phyllis Smith Account number: 711
Street: 1000 Great White Way
City: New York, NY

Old balance: 260.00 Current payment: 0.00 New balance: 260.00

Account status: DELINQUENT

You should understand that this example is unrealistic from a practical standpoint, for two reasons. First, the array of structures (*customer*) is defined to be external to all of the functions within the program. It would be preferable to declare *customer* within *main*, and then pass it to or from *readinput* or *writeoutput* as required. We will learn how to do this in Sec. 11.5.

A more serious problem is the fact that a real customer billing system will store the customer records within a data file on an auxiliary memory device, such as a hard disk or a magnetic tape. To update a record we would access the record from the data file, change the data where necessary, and then write the updated record back to the data file. The use of data files will be discussed in Chap. 12. Since the present example does not make use of data files, we must reenter all of the customer records whenever the program is run. This is rather contrived, though it does provide a simple example illustrating the manner in which structures can be processed on a member-by-member basis.

It is sometimes useful to determine the number of bytes required by an array or a structure. This information can be obtained through the use of the *sizeof* operator, originally discussed in Sec. 3.2. For example, the size of a structure can be determined by writing either *sizeof variable* or *sizeof (struct tag)*.

EXAMPLE 11.15 An elementary C program is shown below.

```
#include <stdio.h>

main() /* determine the size of a structure */
{
    struct date {
        int month;
        int day;
        int year;
    };
}
```

```

struct account {
    int acct_no;
    char acct_type;
    char name[80];
    float balance;
    struct date lastpayment;
} customer;

printf("%d\n", sizeof customer);
printf("%d", sizeof (struct account));
}

```

This program makes use of the `sizeof` operator to determine the number of bytes associated with the structure variable `customer` (or equivalently, the structure `account`). The two `printf` statements illustrate different ways to utilize the `sizeof` operator. Both `printf` statements will produce the same output.

Execution of the program will result in the following output.

93

93

Thus, the structure variable `customer` (or the structure `account`) will occupy 93 bytes. This value is obtained as follows.

| <u>Structure member</u> | <u>Number of bytes</u> |
|--------------------------|------------------------|
| <code>acct_no</code> | 2 |
| <code>acct_type</code> | 1 |
| <code>name</code> | 80 |
| <code>balance</code> | 4 |
| <code>lastpayment</code> | 6 |
| Total | 93 |

Some compilers may assign two bytes to `acct_type` in order to maintain an even number of bytes. Hence, the total byte count may be 94 rather than 93.

11.3 USER-DEFINED DATA TYPES (`typedef`)

The `typedef` feature allows users to define new data-types that are equivalent to existing data types. Once a user-defined data type has been established, then new variables, arrays, structures, etc. can be declared in terms of this new data type.

In general terms, a new data type is defined as

```
typedef type new-type;
```

where `type` refers to an existing data type (either a standard data type, or previous user-defined data type), and `new-type` refers to the new user-defined data type. It should be understood, however, that the new data type will be new in name only. In reality, this new data type will not be fundamentally different from one of the standard data types.

EXAMPLE 11.16 Here is a simple declaration involving the use of `typedef`.

```
typedef int age;
```

In this declaration `age` is a user-defined data type, which is equivalent to type `int`. Hence, the variable declaration

```
age male, female;
```

is equivalent to writing

```
int male, female;
```

In other words, `male` and `female` are regarded as variables of type `age`, though they are actually integer-type variables.

Similarly, the declarations

```
typedef float height[100];
height men, women;
```

define `height` as a 100-element, floating-point array type—hence, `men` and `women` are 100-element, floating-point arrays. Another way to express this is

```
typedef float height;
height men[100], women[100];
```

though the former declaration is somewhat simpler.

The `typedef` feature is particularly convenient when defining structures, since it eliminates the need to repeatedly write `struct tag` whenever a structure is referenced. Hence, the structure can be referenced more concisely. In addition, the name given to a user-defined structure type often suggests the purpose of the structure within the program.

In general terms, a user-defined structure type can be written as

```
typedef struct {
    member 1;
    member 2;
    . . .
    member n;
} new-type;
```

where `new-type` is the user-defined structure type. Structure variables can then be defined in terms of the new data type.

EXAMPLE 11.17 The following declarations are comparable to the structure declarations presented in Examples 11.1 and 11.2. Now, however, we introduce a user-defined data type to describe the structure.

```
typedef struct {
    int acct_no;
    char acct_type;
    char name[80];
    float balance;
} record;

record oldcustomer, newcustomer;
```

The first declaration defines `record` as a user-defined data type. The second declaration defines `oldcustomer` and `newcustomer` as structure variables of type `record`.

The `typedef` feature can be used repeatedly, to define one data type in terms of other user-defined data types.

EXAMPLE 11.18 Here are some variations of the structure declarations presented in Example 11.5.

```

typedef struct {
    int month;
    int day;
    int year;
} date;

typedef struct {
    int acct_no;
    char acct_type;
    char name[80];
    float balance;
    date lastpayment;
} record;

record customer[100];

```

In this example `date` and `record` are user-defined structure types, and `customer` is a 100-element array whose elements are structures of type `record`. (Recall that `date` was a tag rather than an actual data type in Example 11.5.) The individual members within the *i*th element of `customer` can be written as `customer[i].acct_no`, `customer[i].name`, `customer[i].lastpayment.month`, etc., as before.

There are, of course, variations on this theme. Thus, an alternate declaration can be written as

```

typedef struct {
    int month;
    int day;
    int year;
} date;

typedef struct {
    int acct_no;
    char acct_type;
    char name[80];
    float balance;
    date lastpayment;
} record[100];

record customer;

```

or simply

```

typedef struct {
    int month;
    int day;
    int year;
} date;

struct {
    int acct_no;
    char acct_type;
    char name[80];
    float balance;
    date lastpayment;
} customer[100];

```

All three sets of declarations are equivalent.

11.4 STRUCTURES AND POINTERS

The beginning address of a structure can be accessed in the same manner as any other address, through the use of the address (&) operator. Thus, if *variable* represents a structure-type variable, then *&variable* represents the starting address of that variable. Moreover, we can declare a pointer variable for a structure by writing

```
type *ptvar;
```

where *type* is a data type that identifies the composition of the structure, and *ptvar* represents the name of the pointer variable. We can then assign the beginning address of a structure variable to this pointer by writing

```
ptvar = &variable;
```

EXAMPLE 11.19 Consider the following structure declaration, which is a variation of the declaration presented in Example 11.1.

```
typedef struct {
    int acct_no;
    char acct_type;
    char name[80];
    float balance;
} account;

account customer, *pc;
```

In this example *customer* is a structure variable of type *account*, and *pc* is a pointer variable whose object is a structure variable of type *account*. Thus, the beginning address of *customer* can be assigned to *pc* by writing

```
pc = &customer;
```

The variable and pointer declarations can be combined with the structure declaration by writing

```
struct {
    member 1;
    member 2;
    . . . .
    member m;
} variable, *ptvar;
```

where *variable* again represents a structure-type variable, and *ptvar* represents the name of a pointer variable.

EXAMPLE 11.20 The following single declaration is equivalent to the two declarations presented in the previous example.

```
struct {
    int acct_no;
    char acct_type;
    char name[80];
    float balance;
} customer, *pc;
```

The beginning address of *customer* can be assigned to *pc* by writing

```
pc = &customer;
```

as in the previous example.

An individual structure member can be accessed in terms of its corresponding pointer variable by writing

```
ptvar->member
```

where *ptvar* refers to a structure-type pointer variable and the operator `->` is comparable to the period `(.)` operator discussed in Sec. 11.2. Thus, the expression

```
ptvar->member
```

is equivalent to writing

```
variable.member
```

where *variable* is a structure-type variable, as discussed in Sec. 11.2. The operator `->` falls into the highest precedence group, like the period operator `(.)`. Its associativity is left to right (see Appendix C).

The `->` operator can be combined with the period operator to access a submember within a structure (i.e., to access a member of a structure that is itself a member of another structure). Hence, a submember can be accessed by writing

```
ptvar->member.submember
```

Similarly, the `->` operator can be used to access an element of an array that is a member of a structure. This is accomplished by writing

```
ptvar->member[expression]
```

where *expression* is a nonnegative integer that indicates the array element.

EXAMPLE 11.21 Here is a variation of the declarations shown in Example 11.8.

```
typedef struct {
    int month;
    int day;
    int year;
} date;

struct {
    int acct_no;
    char acct_type;
    char name[80];
    float balance;
    date lastpayment;
} customer, *pc = &customer;
```

Notice that the pointer variable *pc* is initialized by assigning it the beginning address of the structure variable *customer*. In other words, *pc* will point to *customer*.

If we wanted to access the customer's account number, we could write any of the following:

```
customer.acct_no      pc->acct_no      (*pc).acct_no
```

The parentheses are required in the last expression because the period operator has a higher precedence than the indirection operator `(*)`. Without the parentheses the compiler would generate an error, because *pc* (a pointer) is not directly compatible with the dot operator.

Similarly, the customer's balance can be accessed by writing any of the following:

`customer.balance` `pc->balance` `(*pc).balance`

and the month of the last payment can be accessed by writing any of the following:

`customer.lastpayment.month` `pc->lastpayment.month` `(*pc).lastpayment.month`

Finally, the customer's name can be accessed by writing any of the following:

`customer.name` `pc->name` `(*pc).name`

Therefore, the third character of the customer's name can be accessed by writing any of the following (see Sec. 10.4).

`customer.name[2]` `pc->name[2]` `(*pc).name[2]`
`*(customer.name + 2)` `pc->(name + 2)` `*((*pc).name + 2)`

A structure can also include one or more pointers as members. Thus, if `ptmember` is both a pointer and a member of `variable`, then `*variable.ptmember` will access the value to which `ptmember` points. Similarly, if `ptvar` is a pointer variable that points to a structure and `ptmember` is a member of that structure, then `*ptvar->ptmember` will access the value to which `ptmember` points.

EXAMPLE 11.22 Consider the simple C program shown below.

```
#include <stdio.h>

main()
{
    int n = 3333;
    char t = 'C';
    float b = 99.99;

    typedef struct {
        int month;
        int day;
        int year;
    } date;

    struct {
        int *acct_no;
        char *acct_type;
        char *name;
        float *balance;
        date lastpayment;
    } customer, *pc = &customer;

    customer.acct_no = &n;
    customer.acct_type = &t;
    customer.name = "Smith";
    customer.balance = &b;

    printf("%d %c %s %.2f\n", *customer.acct_no, *customer.acct_type,
           customer.name, *customer.balance);
    printf("%d %c %s %.2f", *pc->acct_no, *pc->acct_type,
           pc->name, *pc->balance);
}
```

Within the second structure, the members `acct_no`, `acct_type`, `name` and `balance` are written as pointers. Thus, the value to which `acct_no` points can be accessed by writing either `*customer.acct_no` or `*pc->acct_no`. The same is true for `acct_type` and `balance`. Moreover, recall that a string can be assigned directly to a character-type pointer. Therefore, if `name` points to the beginning of a string, then the string can be accessed by writing either `customer.name` or `pc->name`.

Execution of this simple program results in the following two lines of output.

```
3333 C Smith 99.99
3333 C Smith 99.99
```

The two lines of output are identical, as expected.

Since the `->` operator is a member of the highest precedence group, it will be given the same high priority as the period `(.)` operator, with left-to-right associativity. Moreover, this operator, like the period operator, will take precedence over any unary, arithmetic, relational, logical or assignment operators that may appear in an expression. We have already discussed this point, as it applies to the period operator, in Sec. 11.2. However, some additional consideration should be given to certain unary operators, such as `++`, as they apply to structure-type pointer variables.

We already know that expressions such as `++ptvar->member` and `++ptvar->member.submember` are equivalent to `++(ptvar->member)` and `++(ptvar->member.submember)`, respectively. Thus, such expressions will cause the value of the member or the submember to be incremented, as discussed in Sec. 11.2. On the other hand, the expression `++ptvar` will cause the value of `ptvar` to increase by whatever number of bytes is associated with the structure to which `ptvar` points. (The number of bytes associated with a particular structure can be determined through the use of the `sizeof` operator, as illustrated in Example 11.15.) Hence, the address represented by `ptvar` will change as a result of this expression. Similarly, the expression `(++ptvar).member` will cause the value of `ptvar` to increase by this number of bytes before accessing `member`. There is some danger in attempting operations like these, because `ptvar` may no longer point to a structure variable once its value has been altered.

EXAMPLE 11.23 Here is a variation of the simple C program shown in Example 11.15.

```
#include <stdio.h>

main()
{
    typedef struct {
        int month;
        int day;
        int year;
    } date;

    struct {
        int acct_no;
        char acct_type;
        char name[80];
        float balance;
        date lastpayment;
    } customer, *pt = &customer;

    printf("Number of bytes (dec): %d\n", sizeof *pt);
    printf("Number of bytes (hex): %x\n\n", sizeof *pt);
    printf("Starting address (hex): %x\n", pt);
    printf("Incremented address (hex): %x", ++pt);
}
```

Notice that `pt` is a pointer variable whose object is the structure variable `customer`.

The first `printf` statement causes the number of bytes associated with `customer` to be displayed as a decimal quantity. The second `printf` statement displays this same value as a hexadecimal quantity. The third `printf` statement causes the value of `pt` (i.e., the starting address of `customer`) to be displayed in hexadecimal, whereas the fourth `printf` statement shows what happens when `pt` is incremented.

Execution of the program causes the following output to be generated.

```
Number of bytes (dec): 93
Number of bytes (hex): 5d

Starting address (hex): f72
Incremented address (hex): fc1
```

Thus, we see that `customer` requires 93 decimal bytes, which is 5d in hexadecimal. The initial value assigned to `pt` (i.e., the starting address of `customer`) is f72, in hexadecimal. When `pt` is incremented, its value increases by 5d hexadecimal bytes, to fc1.

It is interesting to alter this program by replacing the character array member `name[80]` with the character pointer `*name`, and then execute the program. What do you think will happen?

11.5 PASSING STRUCTURES TO FUNCTIONS

There are several different ways to pass structure-type information to or from a function. Structure members can be transferred individually, or entire structures can be transferred. The mechanics for carrying out the transfers vary, depending on the type of transfer (individual members or complete structures) and the particular version of C.

Individual structure members can be passed to a function as arguments in the function call, and a single structure member can be returned via the `return` statement. To do so, each structure member is treated the same as an ordinary single-valued variable.

EXAMPLE 11.24 The skeletal outline of a C program is shown below. This outline makes use of the structure declarations presented earlier.

```
float adjust(char name[], int acct_no, float balance);      /* funct prototype */

main()
{
    typedef struct {                                     /* structure declaration */
        int month;
        int day;
        int year;
    } date;

    struct {                                         /* structure declaration */
        int acct_no;
        char acct_type;
        char name[80];
        float balance;
        date lastpayment;
    } customer;

    . . .

    customer.balance = adjust(customer.name, customer.acct_no, customer.balance);

    . . .
}
```

```

float adjust(char name[], int acct_no, float balance)
{
    float newbalance;           /* local variable declaration */

    . . .

    newbalance = . . . . .;      /* adjust value of balance */

    . . .

    return(newbalance);
}

```

This program outline illustrates the manner in which structure members can be passed to a function. In particular, `customer.name`, `customer.acct_no` and `customer.balance` are passed to the function `adjust`. Within `adjust`, the value assigned to `newbalance` presumably makes use of the information passed to the function. This value is then returned to `main`, where it is assigned to the structure member `customer.balance`.

Notice the function declaration in `main`. This declaration could also have been written without the argument names, as follows:

```
float adjust(char [], int, float);
```

Some programmers prefer this form, since it avoids the specification of dummy argument names for data items that are actually structure members. We will continue to utilize full function prototypes, however, to take advantage of the resulting error checking.

A complete structure can be transferred to a function by passing a structure-type pointer as an argument. In principle, this is similar to the procedure used to transfer an array to a function. However, we must use explicit pointer notation to represent a structure that is passed as an argument.

You should understand that a structure passed in this manner will be passed by *reference* rather than by *value*. Hence, if any of the structure members are altered within the function, the alterations will be recognized outside of the function. Again, we see a direct analogy with the transfer of arrays to a function.

EXAMPLE 11.25 Consider the simple C program shown below.

```

#include <stdio.h>

typedef struct {
    char *name;
    int acct_no;
    char acct_type;
    float balance;
} record;

main() /* transfer a structure-type pointer to a function */
{
    void adjust(record *pt); /* function declaration */

    static record customer = {"Smith", 3333, 'C', 33.33};

    printf("%s %d %c %.2f\n", customer.name, customer.acct_no,
           customer.acct_type, customer.balance);

    adjust(&customer);
    printf("%s %d %c %.2f\n", customer.name, customer.acct_no,
           customer.acct_type, customer.balance);
}

```

```

void adjust(record *pt)      /* function definition */

{
    pt->name = "Jones";
    pt->acct_no = 9999;
    pt->acct_type = 'R';
    pt->balance = 99.99;
    return;
}

```

This program illustrates the transfer of a structure to a function by passing the structure's address (a pointer) to the function. In particular, *customer* is a static structure of type *record*, whose members are assigned an initial set of values. These initial values are displayed when the program begins to execute. The structure's address is then passed to the function *adjust*, where different values are assigned to the members of the structure.

Within *adjust*, the formal argument declaration defines *pt* as a pointer to a structure of type *record*. Also, notice the empty *return* statement; i.e., nothing is explicitly returned from *adjust* to *main*.

Within *main*, we see that the current values assigned to the members of *customer* are displayed a second time, after *adjust* has been accessed. Thus, the program illustrates whether or not the changes made in *adjust* carry over to the calling portion of the program.

Execution of the program results in the following output.

```

Smith 3333 C 33.33
Jones 9999 R 99.99

```

Thus, the values assigned to the members of *customer* within *adjust* are recognized within *main*, as expected.

A pointer to a structure can be returned from a function to the calling portion of the program. This feature may be useful when several structures are passed to a function, but only one structure is returned.

EXAMPLE 11.26 Locating Customer Records Here is a simple C program that illustrates how an array of structures is passed to a function, and how a pointer to a particular structure is returned.

Suppose we specify an account number for a particular customer and then locate and display the complete record for that customer. Each customer record will be maintained in a structure, as in the last example. Now, however, the entire set of customer records will be stored in an array called *customer*. Each element of *customer* will be an independent structure.

The basic strategy will be to enter an account number, and then transfer both the account number and the array of records to a function called *search*. Within *search*, the specified account number will be compared with the account number that is stored within each customer record until a match is found, or until the entire list of records has been searched. If a match is found, a pointer to that array element (the structure containing the desired customer record) is returned to *main*, and the contents of the record are displayed.

If a match is not found after searching the entire array, then the function returns a value of *NULL* (zero) to *main*. The program then displays an error message requesting that the user reenter the account number. This overall search procedure will continue until a value of zero is entered for the account number.

The complete program is shown below. Within this program, *customer* is an array of structures of type *record*, and *pt* is a pointer to a structure of this same type. Also, *search* is a function that accepts two arguments and returns a pointer to a structure of type *record*. The arguments are an array of structures of type *record* and an integer quantity, respectively. Within *search*, the quantity returned is either the address of an array element, or *NULL* (zero).

```

/* find a customer record that corresponds to a specified account number */

#include <stdio.h>

#define N 3
#define NULL 0

```

```
typedef struct {
    char *name;
    int acct_no;
    char acct_type;
    float balance;
} record;

record *search(record table[], int acctn); /* function prototype */

main()
{
    static record customer[N] = {
        {"Smith", 3333, 'C', 33.33},
        {"Jones", 6666, 'O', 66.66},
        {"Brown", 9999, 'D', 99.99}
    }; /* array of structures */

    int acctn; /* variable declaration */
    record *pt; /* pointer declaration */

    printf("Customer Account Locator\n");
    printf("To END, enter 0 for the account number\n");
    printf("\nAccount no.: "); /* enter first account number */
    scanf("%d", &acctn);

    while (acctn != 0) {
        pt = search(customer, acctn);

        if (pt != NULL) /* found a match */
            printf("\nName: %s\n", pt->name);
            printf("Account no.: %d\n", pt->acct_no);
            printf("Account type: %c\n", pt->acct_type);
            printf("Balance: %.2f\n", pt->balance);
        }
        else
            printf("\nERROR - Please try again\n");

        printf("\nAccount no.: "); /* enter next account number */
        scanf("%d", &acctn);
    }
}

record *search(record table[N], int acctn) /* function definition */
/* accept an array of structures and an account number,
   return a pointer to a particular structure (an array element)
   if the account number matches a member of that structure */

{
    int count;

    for (count = 0; count < N; ++count)
        if (table[count].acct_no == acctn) /* found a match */
            return(&table[count]); /* return pointer to array element */

    return(NULL);
}
```

The array size is expressed in terms of the symbolic constant N. For this simple example we have selected a value of N = 3. That is, we are storing only three sample records within the array. In a more realistic example, N would have a much greater value.

Finally, it should be mentioned that there are much better ways to search through a set of records than examining each record sequentially. We have selected this simple though inefficient procedure in order to concentrate on the mechanics of transferring structures between `main` and its subordinate function `search`.

Shown below is a typical dialog that might result from execution of the program. The user's responses are underlined, as usual.

```
Customer Account Locator
To END, enter 0 for the account number

Account no.: 3333

Name: Smith
Account no.: 3333
Account type: C
Balance: 33.33

Account no.: 9999

Name: Brown
Account no.: 9999
Account type: D
Balance: 99.99

Account no.: 666

ERROR - Please try again

Account no.: 6666

Name: Jones
Account no.: 6666
Account type: O
Balance: 66.66

Account no.: 0
```

Newer versions of C permit an entire structure to be transferred directly to a function as an argument, and returned directly from a function via the `return` statement. (Notice the contrast with arrays, which *cannot* be returned via the `return` statement.) These features are included in the current ANSI standard.

When a structure is passed directly to a function, the transfer is by value rather than by reference. This is consistent with other direct (nonpointer) transfers in C. Therefore, if any of the structure members are altered within the function, the alterations *will not* be recognized outside of the function. However, if the altered structure is returned to the calling portion of the program, then the changes *will* be recognized within this broader scope.

EXAMPLE 11.27 In Example 11.25 we saw a program that transferred a structure-type pointer to a function. Two different `printf` statements within `main` illustrated the fact that transfers of this type are by reference; i.e., alterations made to the structure within the function are recognized within `main`. A similar program is shown below. However, the present program transfers a complete structure, rather than a structure-type pointer, to the function.

```

#include <stdio.h>

typedef struct {
    char *name;
    int acct_no;
    char acct_type;
    float balance;
} record;

void adjust(record customer);           /* function prototype */

main()      /* transfer a structure to a function */
{
    static record customer = {"Smith", 3333, 'C', 33.33};

    printf("%s %d %c %.2f\n", customer.name, customer.acct_no,
           customer.acct_type, customer.balance);

    adjust(customer);
    printf("%s %d %c %.2f\n", customer.name, customer.acct_no,
           customer.acct_type, customer.balance);
}

void adjust(record cust)           /* function definition */
{
    cust.name = "Jones";
    cust.acct_no = 9999;
    cust.acct_type = 'R';
    cust.balance = 99.99;
    return;
}

```

Notice that the function **adjust** now accepts a structure of type **record** as an argument, rather than a pointer to a structure of type **record**, as in Example 11.25. Nothing is returned from **adjust** to **main** in either program.

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

```

Smith 3333 C 33.33
Smith 3333 C 33.33

```

Thus, the new assignments made within **adjust** are not recognized within **main**. This is expected, since the transfer of the structure **customer** from **main** to **adjust** is by value rather than by reference. (Compare with the output shown in Example 11.25.)

Now suppose we modify this program so that the altered structure is returned from **adjust** to **main**. Here is the modified program.

```

#include <stdio.h>

typedef struct {
    char *name;
    int acct_no;
    char acct_type;
    float balance;
} record;

record adjust(record customer);           /* function prototype */

```

```

main() /* transfer a structure to a function and return the structure */

{
    static record customer = {"Smith", 3333, 'C', 33.33};

    printf("%s %d %c %.2f\n", customer.name, customer.acct_no,
           customer.acct_type, customer.balance);

    customer = adjust(customer);
    printf("%s %d %c %.2f\n", customer.name, customer.acct_no,
           customer.acct_type, customer.balance);
}

record adjust(record cust) /* function definition */

{
    cust.name = "Jones";
    cust.acct_no = 9999;
    cust.acct_type = 'R';
    cust.balance = 99.99;
    return(cust);
}

```

Notice that **adjust** now returns a structure of type **record** to **main**. The **return** statement is modified accordingly. Execution of this program results in the following output.

```

Smith 3333 C 33.33
Jones 9999 R 99.99

```

Thus, the alterations that were made within **adjust** are now recognized within **main**. This is expected, since the altered structure is now returned directly to the calling portion of the program. (Compare with the output shown in Example 11.25 as well as the output shown earlier in this example.)

Most versions of C allow complicated data structures to be transferred freely between functions. We have already seen examples involving the transfer of individual structure members, entire structures, pointers to structures and arrays of structures. As a practical matter, however, there are some limitations on the complexity of data structures that can easily be transferred to or from a function. In particular, some compilers may have difficulty executing programs that involve complex data structure transfers, because of certain memory restrictions. Beginning programmers should be aware of these limitations, though the details of this topic are beyond the scope of the current text.

EXAMPLE 11.28 Updating Customer Records Example 11.14 presented a simple customer billing system illustrating the use of structures to maintain and update customer records. In that example the customer records were stored within a global (external) array of structures. We now consider two variations of that program. In each program the array of structures is maintained locally, within **main**. The individual array elements (i.e., individual customer records) are transferred back and forth between functions, as required.

In the first program, complete structures are transferred between the functions. In particular, the function **readinput** allows information defining each customer record to be entered into the computer. When an entire record has been entered, the corresponding structure is returned to **main**, where it is stored within the 100-element array called **customer** and adjusted for the proper account type. After all the records have been entered and adjusted, they are transferred individually to the function **writeoutput**, where certain information is displayed for each customer.

The entire program is shown below.

```
/* update a series of customer accounts (simplified billing system) */

#include <stdio.h>

/* maintain the customer accounts as an array of structures,
   transfer complete structures to and from functions */
```

```
typedef struct {
    int month;
    int day;
    int year;
} date;
```

```
typedef struct {
    char name[80];
    char street[80];
    char city[80];
    int acct_no;           /* (positive integer) */
    char acct_type;        /* C (current), O (overdue), or D (delinquent) */
    float oldbalance;      /* (nonnegative quantity) */
    float newbalance;      /* (nonnegative quantity) */
    float payment;         /* (nonnegative quantity) */
    date lastpayment;
} record;
```

```
record readinput(int i);          /* function prototype */
void writeoutput(record customer); /* function prototype */

main()
/* read customer accounts, process each account, and display output */

{
    int i, n;                  /* variable declarations */
    record customer[100];       /* array declaration (array of structures) */

    printf("CUSTOMER BILLING SYSTEM\n\n");
    printf("How many customers are there? ");
    scanf("%d", &n);

    for (i = 0; i < n; ++i)    {
        customer[i] = readinput(i);

        /* determine account status */

        if (customer[i].payment > 0)
            customer[i].acct_type =
                (customer[i].payment < 0.1 * customer[i].oldbalance) ? 'O' : 'C';
        else
            customer[i].acct_type =
                (customer[i].oldbalance > 0) ? 'D' : 'C';

        /* adjust account balance */

        customer[i].newbalance = customer[i].oldbalance - customer[i].payment;
    }

    for (i = 0; i < n; ++i)
        writeoutput(customer[i]);
}
```

```
record readinput(int i)
/* read input data for a customer */

{
    record customer;

    printf("\nCustomer no. %d\n", i + 1);
    printf("  Name: ");
    scanf(" %[^\n]", customer.name);
    printf("  Street: ");
    scanf(" %[^\n]", customer.street);
    printf("  City: ");
    scanf(" %[^\n]", customer.city);
    printf("  Account number: ");
    scanf("%d", &customer.acct_no);
    printf("  Previous balance: ");
    scanf("%f", &customer.oldbalance);
    printf("  Current payment: ");
    scanf("%f", &customer.payment);
    printf("  Payment date (mm/dd/yyyy): ");
    scanf("%d/%d/%d", &customer.lastpayment.month,
          &customer.lastpayment.day,
          &customer.lastpayment.year);

    return(customer);
}

void writeoutput(record customer)
/* display current information for a customer */

{
    printf("\nName:  %s", customer.name);
    printf("  Account number: %d\n", customer.acct_no);
    printf("Street: %s\n", customer.street);
    printf("City:  %s\n\n", customer.city);
    printf("Old balance: %7.2f", customer.oldbalance);
    printf("  Current payment: %7.2f", customer.payment);
    printf("  New balance: %7.2f\n\n", customer.newbalance);
    printf("Account status: ");

    switch (customer.acct_type)  {
        case 'C':
            printf("CURRENT\n\n");
            break;
        case 'O':
            printf("OVERDUE\n\n");
            break;
        case 'D':
            printf("DELINQUENT\n\n");
            break;
        default:
            printf("ERROR\n\n");
    }
    return;
}
```

The next program is very similar to the previous program. Now, however, the transfers involve pointers to structures rather than the structures themselves. Thus, the structures are now transferred by reference, whereas they were transferred by value in the previous program.

For brevity, this program is outlined rather than listed in its entirety. The missing blocks are identical to the corresponding portions of the previous program.

```
/* update a series of customer accounts (simplified billing system) */

#include <stdio.h>

/* maintain the customer accounts as an array of structures,
   transfer pointers to structures to and from functions      */

/* (structure definitions) */

record *readinput(int i);           /* function prototype */
void writeoutput(record *cust);    /* function prototype */

main()
{
    /* read customer accounts, process each account, and display output */

    {
        int i, n;                  /* variable declarations */
        record customer[100];     /* array declaration (array of structures) */

        . . .

        for (i = 0; i < n; ++i)  {
            customer[i] = *readinput(i);

            /* determine account status */

            . . .

            /* adjust account balance */

            . . .
        }

        for (i = 0; i < n; ++i)
            writeoutput(&customer[i]);
    }

    record *readinput(int i)
    /* read input data for a customer */
    {
        record customer;

        /* enter input data */

        return(&customer);
    }
}
```

```

void writeoutput(record *pt)
/* display current information for a customer */

{
    record customer;
    customer = *pt;

    /* display output data */

    return;
}

```

Both of these programs will behave in the same manner as the program given in Example 11.14 when executed. Because of the complexity of the data structure (i.e., the array of structures, where each structure contains embedded arrays and embedded structures), however, the compiled programs may not be executable with certain compilers. In particular, a stack overflow condition (a type of inadequate memory condition) may be experienced with some compilers.

This problem would not exist if the program were more realistic; i.e., if the customer records were stored within a file on an auxiliary memory device, rather than in an array that is stored within the computer's memory. We will discuss this problem in Chap. 12, where we consider the use of data files for situations such as this.

11.6 SELF-REFERENTIAL STRUCTURES

It is sometimes desirable to include within a structure one member that is a pointer to the parent structure type. In general terms, this can be expressed as

```

struct tag {
    member 1;
    member 2;
    . . .
    struct tag *name;
};

```

where *name* refers to the name of a pointer variable. Thus, the structure of type *tag* will contain a member that points to another structure of type *tag*. Such structures are known as *self-referential* structures.

EXAMPLE 11.29 A C program contains the following structure declaration.

```

struct list_element {
    char item[40];
    struct list_element *next;
};

```

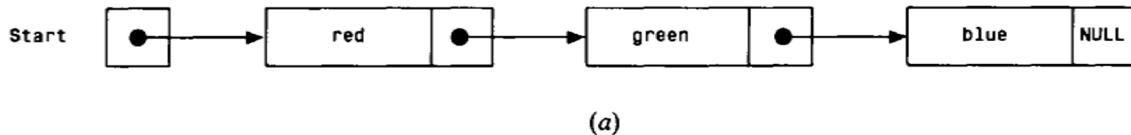
This is a structure of type *list_element*. The structure contains two members: a 40-element character array, called *item*, and a pointer to a structure of the same type (i.e., a pointer to a structure of type *list_element*), called *next*. Therefore this is a self-referential structure.

Self-referential structures are very useful in applications that involve linked data structures, such as lists and trees. We will see a comprehensive example illustrating the processing of a linked list in Example 11.32. First, however, we present a brief summary of linked data structures.

The basic idea of a linked data structure is that each component within the structure includes a pointer indicating where the next component can be found. Therefore, the relative order of the components can easily be changed simply by altering the pointers. In addition, individual components can easily be added or deleted,

again by altering the pointers. As a result, a linked data structure is not confined to some maximum number of components. Rather, the data structure can expand or contract in size as required.

EXAMPLE 11.30 Figure 11.3(a) illustrates a linked list containing three components. Each component consists of two data items: a string, and a pointer that references the next component within the list. Thus, the first component contains the string **red**, the second contains **green** and the third contains **blue**. The beginning of the list is indicated by a separate pointer, which is labeled **start**. Also, the end of the list is indicated by a special pointer, called **NULL**.



Now let us add another component, whose value is **white**, between **red** and **green**. To do so we merely change the pointers, as illustrated in Fig. 11.3(b). Similarly, if we choose to delete the component whose value is **green**, we simply change the pointer associated with the second component, as shown in Fig. 11.3(c).

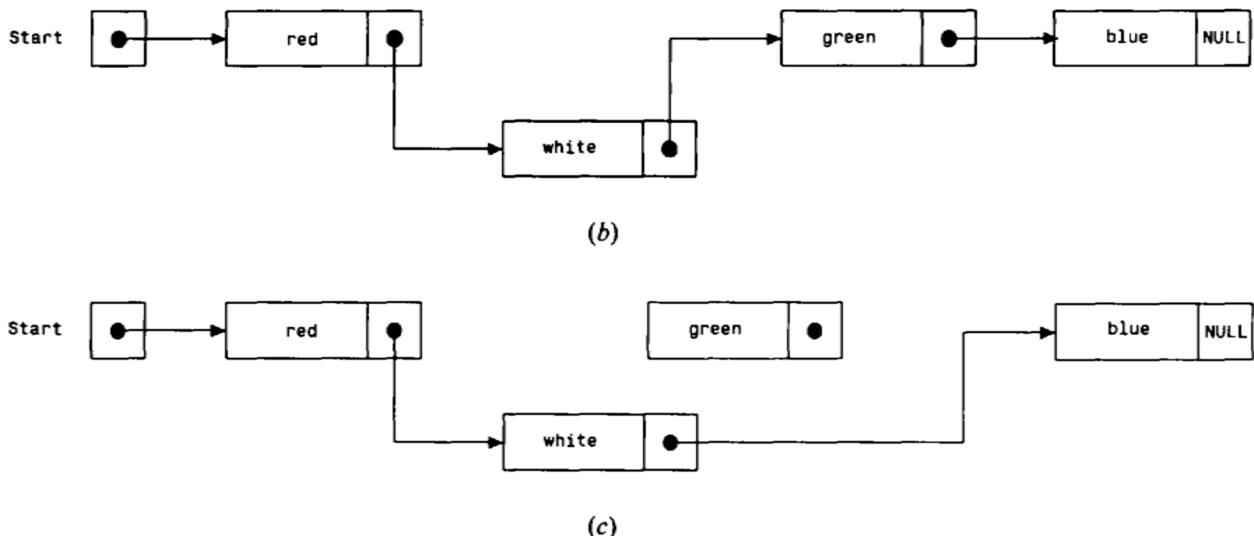


Fig. 11.3

There are several different kinds of linked data structures, including *linear* linked lists, in which the components are all linked together sequentially; linked lists with *multiple pointers*, which permit forward and backward traversal within the list; *circular* linked lists, which have no beginning and no ending; and *trees*, in which the components are arranged in a hierarchical structure. We have already seen an illustration of a linear linked list in Example 11.30. Other kinds of linked lists are illustrated in the next example.

EXAMPLE 11.31 Figure 11.4 shows a linear linked list that is similar to that shown in Fig. 11.3(a). Now, however, we see that there are *two* pointers associated with each component: a forward pointer, and a backward pointer. This double set of pointers allows us to traverse the list in either direction, i.e., from beginning to end, or from end to beginning.

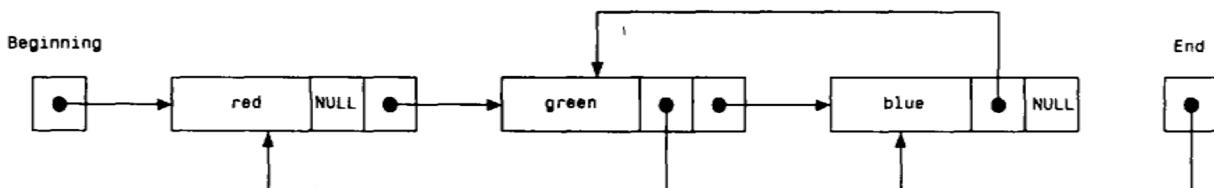


Fig. 11.4

Now consider the list shown in Fig. 11.5. This list is similar to that shown in Fig. 11.3(a), except that the last data item (**blue**) points to the first data item (**red**). Hence, this list has no beginning and no ending. Such lists are referred to as *circular lists*.

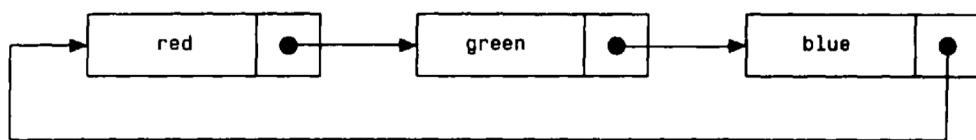


Fig. 11.5

Finally, in Fig. 11.6(a) we see an example of a *tree*. Trees consist of nodes and branches, arranged in some hierarchical manner which indicates a corresponding hierarchical structure within the data. (A *binary tree* is a tree in which every node has no more than two branches.)

In Fig. 11.6(a) the *root node* has the value **screen**, and the associated branches lead to the nodes whose values are **foreground** and **background**, respectively. Similarly, the branches associated with **foreground** lead to the nodes whose values are **white**, **green** and **amber**, and the branches associated with **background** lead to the nodes whose values are **black**, **blue** and **white**.

Figure 11.6(b) illustrates the manner in which pointers are used to construct the tree.

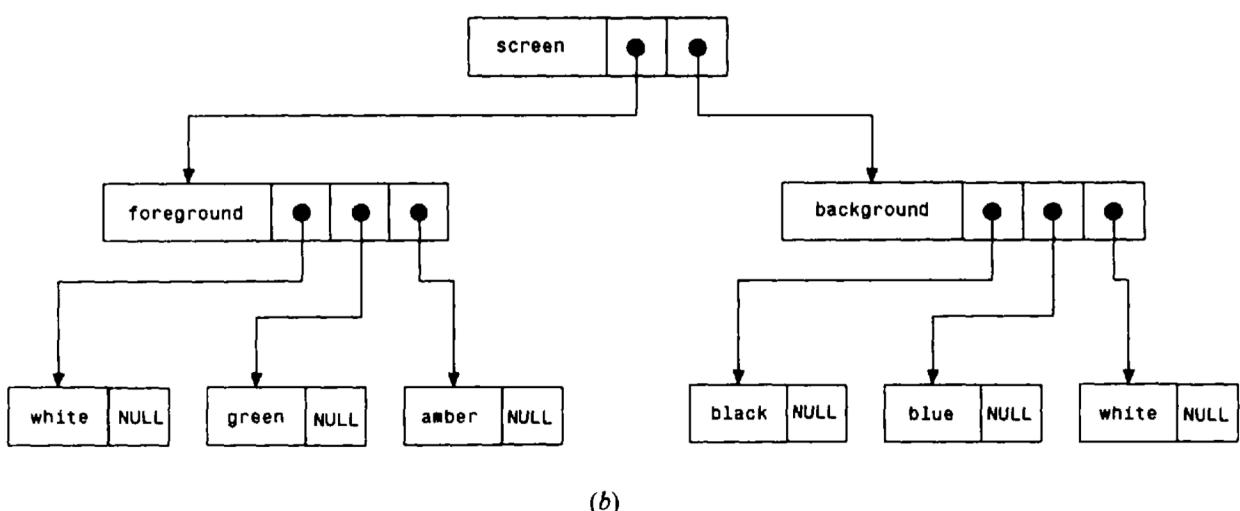
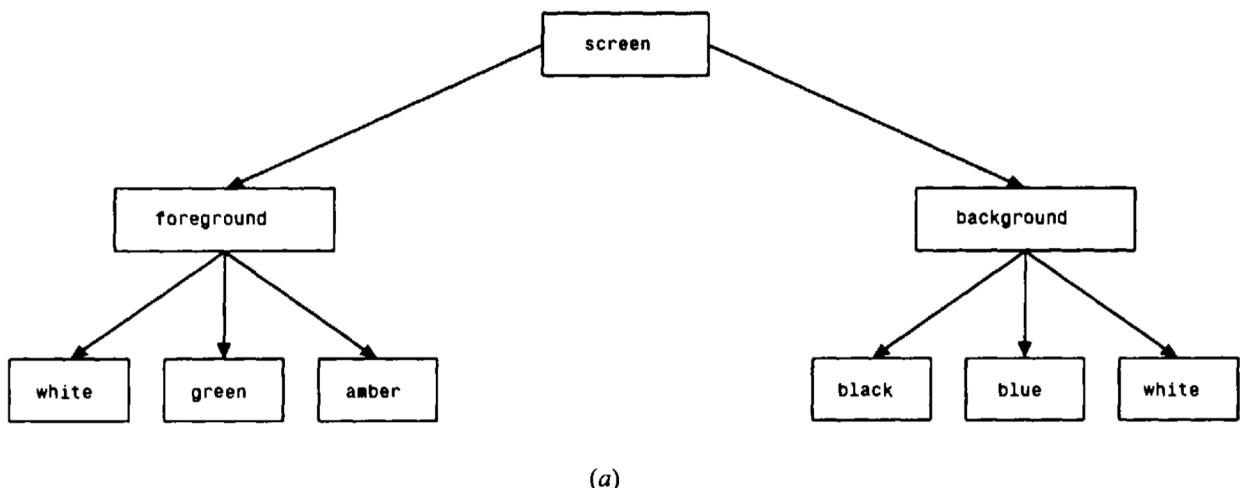


Fig. 11.6

Self-referential structures are ideally suited for applications involving linked data structures. Each structure will represent a single component (i.e., one node) within the linked data structure. The self-referential pointer will indicate the location of the next component.

EXAMPLE 11.32 Processing a Linked List We now present an interactive C program that allows us to create a linear linked list, add a new component to the linked list, or delete an existing component from the linked list. Each component will consist of a string, and a pointer to the next component. The program will be menu-driven to facilitate its use by nonprogrammers. We will include a provision to display the list after the selection of any menu item (i.e., after any change has been made to the list).

This program is somewhat more complex than the preceding example programs. It utilizes both *recursion* (see Sec. 7.6) and *dynamic memory allocation* (see Sec. 10.5, and Examples 10.15, 10.22, 10.24 and 10.26).

The entire program is shown below. Following the program listing, the individual functions are discussed in detail.

```
/* menu-driven program to process a linked list of strings */

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

#define NULL 0

struct list_element {
    char item[40];                      /* data item for this node */
    struct list_element *next;            /* pointer to the next node */
};

typedef struct list_element node;        /* structure type declaration */

int menu(void);                         /* function prototype */
void create(node *pt);                  /* function prototype */
node *insert(node *pt);                 /* function prototype */
node *remove(node *pt);                 /* function prototype */
void display(node *pt);                 /* function prototype */

main()
{
    node *start;                        /* pointer to beginning of list*/
    int choice;                         /* local variable declaration */

    do {
        choice = menu();
        switch (choice) {
            case 1:      /* create the linked list */
                start = (node *) malloc(sizeof(node)); /* allocate space, 1st node */
                create(start);
                printf("\n");
                display(start);
                continue;

            case 2:      /* add one component */
                start = insert(start);
                printf("\n");
                display(start);
                continue;
        }
    }
}
```

```
    case 3:      /* delete one component */
        start = remove(start);
        printf("\n");
        display(start);
        continue;

    default:     /* terminate computation */
        printf("End of computation\n");
    }
} while (choice != 4);
}

int menu(void)      /* generate the main menu */
{
    int choice;

    do  {
        printf("\nMain menu:\n");
        printf(" 1 - CREATE the linked list\n");
        printf(" 2 - ADD a component\n");
        printf(" 3 - DELETE a component\n");
        printf(" 4 - END\n");
        printf("Please enter your choice (1, 2, 3 or 4) -> ");
        scanf("%d", &choice);
        if (choice < 1 || choice > 4)
            printf("\nERROR - Please try again\n");
    } while (choice < 1 || choice > 4);
    printf("\n");
    return(choice);
}

void create(node *record)      /* create a linked list */
/* argument points to the current node */
{
    printf("Data item (type 'END' when finished): ");
    scanf(" %[^\n]", record->item);

    if (strcmp(record->item, "END") == 0)
        record->next = NULL;
    else  {
        /* allocate space for next node */
        record->next = (node *) malloc(sizeof(node));

        /* create the next node */
        create(record->next);
    }
    return;
}

void display(node *record)      /* display the linked list */
/* argument points to the current node */
```

```
{  
    if (record->next != NULL) {  
        printf("%s\n", record->item); /* display this data item */  
        display(record->next); /* get the next data item */  
    }  
    return;  
}  
  
node *insert(node *first) /* add one component to the linked list  
                           return a pointer to beginning of the modified list */  
/* argument points to the first node */  
  
{  
    node *locate(node*, char[]); /* function declaration */  
    node *newrecord; /* pointer to new node */  
    node *tag; /* pointer to node BEFORE target node */  
    char newitem[40]; /* new data item */  
    char target[40]; /* data item following the new entry */  
  
    printf("New data item: ");  
    scanf(" %[^\n]", newitem);  
    printf("Place before (type '\\END\\' if last): ");  
    scanf(" %[^\n]", target);  
  
    if (strcmp(first->item, target) == 0) {  
        /* new node is first in list */  
  
        /* allocate space for the new node */  
        newrecord = (node *) malloc(sizeof(node));  
  
        /* assign the new data item to newrecord->item */  
        strcpy(newrecord->item, newitem);  
  
        /* assign the current pointer to newrecord->next */  
        newrecord->next = first;  
  
        /* new pointer becomes the beginning of the list */  
        first = newrecord;  
    }  
  
    else {  
        /* insert new node after an existing node */  
  
        /* locate the node PRECEDING the target node */  
        tag = locate(first, target);  
  
        if (tag == NULL)  
            printf("\nMatch not found - Please try again\n");  
        else {  
            /* allocate space for the new node */  
            newrecord = (node *) malloc(sizeof(node));  
  
            /* assign the new data item to newrecord->item */  
            strcpy(newrecord->item, newitem);  
  
            /* assign the next pointer to newrecord->next */  
            newrecord->next = tag->next;  
        }  
    }  
}
```

```

        /* assign the new pointer to tag->next */
        tag->next = newrecord;
    }
}
return(first);
}

node *locate(node *record, char target[])      /* locate a node */

/* return a pointer to the node BEFORE the target node
   The first argument points to the current node
   The second argument is the target string          */

{
    if (strcmp(record->next->item, target) == 0)      /* found a match */
        return(record);
    else
        if (record->next->next == NULL)                /* end of list */
            return(NULL);
        else
            locate(record->next, target);              /* try next node */
}

node *remove(node *first)      /* remove (delete) one component from the linked list
                                return a pointer to beginning of the modified list */

/* argument points to the first node */

{
    node *locate(node*, char[]);           /* function declaration */
    node *tag;                          /* pointer to node BEFORE target node */
    node *temp;                         /* temporary pointer */
    char target[40];                    /* data item to be deleted */

    printf("Data item to be deleted: ");
    scanf(" %[^\n]", target);

    if (strcmp(first->item, target) == 0)  {
        /* delete the first node */

        /* mark the node following the target node */
        temp = first->next;

        /* free space for the target node */
        free(first);

        /* adjust the pointer to the first node */
        first = temp;
    }

    else  {
        /* delete a data item other than the first */

        /* locate the node PRECEDING the target node */
        tag = locate(first, target);

        if (tag == NULL)
            printf("\nMatch not found - Please try again\n");
    }
}

```

```

    else {
        /* mark the node following the target node */
        temp = tag->next->next;

        /* free space for the target node */
        free(tag->next);

        /* adjust the link to the next node */
        tag->next = temp;
    }
}

return(first);
}

```

The program begins with the usual `#include` statements and a definition of the symbolic constant `NULL` to represent the value 0. Following these statements is a declaration for the self-referential structure `list_element`. This structure declaration is the same as that shown in Example 11.29. Thus, `list_element` identifies a structure consisting of two members: a 40-element character array (`item`), and a pointer (`next`) to another structure of the same type. The character array will represent a string, and the pointer will identify the location of the next component in the linked list.

The data type `node` is then defined, identifying structures having composition `list_element`. This definition is followed by the function prototypes. Within the function prototypes, notice that `start` is a pointer to a structure of type `node`. This pointer will indicate the beginning of the linked list. The remaining function prototypes identify several additional functions that are called from `main`. Note that these declarations and function prototypes are external. They will therefore be recognized throughout the program.

The `main` function consists of a `do - while` loop that permits repetitious execution of the entire process. This loop calls the function `menu`, which generates the main menu, and returns a value for `choice`, indicating the user's menu selection. A `switch` statement then calls the appropriate functions, in accordance with the user's selection. Notice that the program will stop executing if `choice` is assigned a value of 4.

If `choice` is assigned a value of 1, indicating that a new linked list will be created, a block of memory must be allocated for the first data item before calling the function `create`. This is accomplished using the library function `malloc`, as discussed in Sec. 10.5. Thus, memory allocation statement

```
start = (node *) malloc(sizeof(node));
```

reserves a block of memory whose size (in bytes) is sufficient for one node. The statement returns a pointer to a structure of type `node`. This pointer indicates the beginning of the linked list. Thus, it is passed to `create` as an argument.

Note that the type cast (`node *`) is required as a part of the memory allocation statement. Without it, the `malloc` function would return a pointer to a `char` rather than a pointer to a structure of type `node`.

Now consider the function `menu`, which is used to generate the main menu. This function accepts a value for `choice` after the menu has been generated. The only permissible values for `choice` are 1, 2, 3 or 4. An error trap, in the form of a `do - while` statement, causes an error message to be displayed and a new menu to be generated if a value other than 1, 2, 3 or 4 is entered in response to the menu.

The linked list is created by the function `create`. This is a recursive function that accepts a pointer to the current node (i.e., the node that is being created) as an argument. The pointer variable is called `record`.

The `create` function begins by prompting for the current data item; i.e., the string that is to reside in the current node. If the user enters the string `END` (in either upper- or lowercase), then `NULL` is assigned to the pointer that indicates the location of the next node and the recursion stops. If the user enters any string other than `END`, however, memory is allocated for the next node via the `malloc` function and the function calls itself recursively. Thus, the recursion will continue until the user has entered `END` for one of the data items.

Once the linked list has been created, it is displayed via the function `display`. This function is called from `main`, after the call to `create`. Notice that `display` accepts a pointer to the current node as an argument. The function then executes recursively, until it receives a pointer whose value is `NULL`. The recursion therefore causes the entire linked list to be displayed.

Now consider the function `insert`, which is used to add a new component (i.e., a new node) to the linked list. This function asks the user where the insertion is to occur. Note that the function accepts a pointer to the beginning of the list as an argument, and then returns a pointer to the beginning of the list, after the insertion has been made. These two pointers will be the same, unless the insertion is made at the beginning of the list.

The `insert` function does not execute recursively. It first prompts for the new data item (`newitem`), followed by a prompt for the existing data item that will follow the new data item (the existing data item is called `target`). If the insertion is to be made at the *beginning of the list*, then memory is allocated for the new node, `newitem` is assigned to the first member, and the pointer originally indicating the beginning of the linked list (`first`) is assigned to the second member. The pointer returned by `malloc`, which indicates the beginning of the new node, is then assigned to `first`. Hence, the beginning of the new node becomes the beginning of the entire list.

If the insertion is to be made *after an existing node*, then function `locate` is called to determine the location of the insertion. This function returns a pointer to the node *preceding* the target node. The value returned is assigned to the pointer `tag`. Hence, `tag` points to the node that will precede the new node. If `locate` cannot find a match between the value entered for `target` and an existing data item, it will return `NULL`.

If a match is found by `locate`, then the insertion is made in the following manner: memory is allocated for the new node, `newitem` is assigned to the first member of `newrecord` (i.e., `tonewrecord->item`), and the pointer to the target node (i.e., `tag->next`) is assigned to the second member of `newrecord` (i.e., `newrecord->next`). The pointer returned by `malloc`, which indicates the beginning of the new node, is then assigned to `tag->next`. Hence, the pointer in the preceding node will point to the new node, and the pointer in the new node will point to the target node.

Now consider the function `locate`. This is a simple recursive function that accepts a pointer to the current node and the target string as arguments, and returns a pointer to the node that *precedes* the current node. Therefore, if the data item in the node following the current node matches the target string, the function will return the pointer to the current node. Otherwise, one of two possible actions will be taken. If the pointer in the node following the current node is `NULL`, indicating the end of the linked list, a match has not been found. Therefore, the function will return `NULL`. But, if the pointer in the node following the current node is something other than `NULL`, the function will call itself recursively, thus testing the next node for a match.

Finally, consider the function `remove`, which is used to delete an existing component (i.e., an existing node) from the linked list. This function is similar to `insert`, though somewhat simpler. It accepts a pointer to the beginning of the linked list as an argument, and returns a pointer to the beginning of the linked list after the deletion has been made.

The `remove` function begins by prompting for the data item to be deleted (`target`). If this is the first data item, then the pointers are adjusted as follows: The pointer indicating the location of the second node is temporarily assigned to the pointer variable `temp`; the memory utilized by the first node is freed, using the library function `free`; and the location of the second node (which is now the first node, because of the deletion) is assigned to `first`. Hence, the beginning of the (former) second node becomes the beginning of the entire list.

If the data item to be deleted is *not* the first data item in the list, then `locate` is called to determine the location of the deletion. This function will return a pointer to the node *preceding* the target node. The value returned is assigned to the pointer variable `tag`. If this value is `NULL`, a match cannot be found. An error message is then generated, requesting that the user try again.

If `locate` returns a value other than `NULL`, the target node is deleted in the following manner: The pointer to the node following the target node is temporarily assigned to the pointer variable `temp`; the memory utilized by the target node is then freed, using the library function `free`; and the value of `temp` is then assigned to `tag->next`. Hence, the pointer in the preceding node will point to the node following the target node.

Let us now utilize this program to create a linked list containing the following cities: Boston, Chicago, Denver, New York, Pittsburgh, San Francisco. We will then add several cities and delete several cities, thus illustrating all of the program's features. We will maintain the list of cities in alphabetical order throughout the exercise. (We could, of course, have the computer do the sorting for us, though this would further complicate an already complex program.)

The entire interactive session is shown below. As usual, the user's responses have been underlined.

Main menu:

- 1 - CREATE the linked list
- 2 - ADD a component
- 3 - DELETE a component
- 4 - END

Please enter your choice (1, 2, 3 or 4) -> 1

```
Data item (type 'END' when finished): BOSTON
Data item (type 'END' when finished): CHICAGO
Data item (type 'END' when finished): DENVER
Data item (type 'END' when finished): NEW YORK
Data item (type 'END' when finished): PITTSBURGH
Data item (type 'END' when finished): SAN FRANCISCO
Data item (type 'END' when finished): END
```

BOSTON
CHICAGO
DENVER
NEW YORK
PITTSBURGH
SAN FRANCISCO

Main menu:

- 1 - CREATE the linked list
- 2 - ADD a component
- 3 - DELETE a component
- 4 - END

Please enter your choice (1, 2, 3 or 4) -> 2

New data item: ATLANTA
Place before (type 'END' if last): BOSTON

ATLANTA
BOSTON
CHICAGO
DENVER
NEW YORK
PITTSBURGH
SAN FRANCISCO

Main menu:

- 1 - CREATE the linked list
- 2 - ADD a component
- 3 - DELETE a component
- 4 - END

Please enter your choice (1, 2, 3 or 4) -> 2

New data item: SEATTLE
Place before (type 'END' if last): END

ATLANTA
BOSTON
CHICAGO
DENVER
NEW YORK
PITTSBURGH
SAN FRANCISCO
SEATTLE

Main menu:

- 1 - CREATE the linked list
- 2 - ADD a component
- 3 - DELETE a component
- 4 - END

Please enter your choice (1, 2, 3 or 4) -> 3

Data item to be deleted: NEW YORK

ATLANTA
BOSTON
CHICAGO
DENVER
PITTSBURGH
SAN FRANCISCO
SEATTLE

Main menu:

- 1 - CREATE the linked list
- 2 - ADD a component
- 3 - DELETE a component
- 4 - END

Please enter your choice (1, 2, 3 or 4) -> 2

New data item: WASHINGTON

Place before (type 'END' if last): WILLIAMSBURG

Match not found - Please try again

ATLANTA
BOSTON
CHICAGO
DENVER
PITTSBURGH
SAN FRANCISCO
SEATTLE

Main menu:

- 1 - CREATE the linked list
- 2 - ADD a component
- 3 - DELETE a component
- 4 - END

Please enter your choice (1, 2, 3 or 4) -> 2

New data item: WASHINGTON

Place before (type 'END' if last): END

ATLANTA
BOSTON
CHICAGO
DENVER
PITTSBURGH
SAN FRANCISCO

SEATTLE
WASHINGTON

Main menu:

- 1 - CREATE the linked list
- 2 - ADD a component
- 3 - DELETE a component
- 4 - END

Please enter your choice (1, 2, 3 or 4) -> 3

Data item to be deleted: ATLANTA

BOSTON
CHICAGO
DENVER
PITTSBURGH
SAN FRANCISCO
SEATTLE
WASHINGTON

Main menu:

- 1 - CREATE the linked list
- 2 - ADD a component
- 3 - DELETE a component
- 4 - END

Please enter your choice (1, 2, 3 or 4) -> 2

New data item: DALLAS

Place before (type 'END' if last): DENVER

BOSTON
CHICAGO
DALLAS
DENVER
PITTSBURGH
SAN FRANCISCO
SEATTLE
WASHINGTON

Main menu:

- 1 - CREATE the linked list
- 2 - ADD a component
- 3 - DELETE a component
- 4 - END

Please enter your choice (1, 2, 3 or 4) -> 3

Data item to be deleted: MIAMI

Match not found - Please try again

BOSTON
CHICAGO
DALLAS

```
DENVER  
PITTSBURGH  
SAN FRANCISCO  
SEATTLE  
WASHINGTON
```

Main menu:

- 1 - CREATE the linked list
- 2 - ADD a component
- 3 - DELETE a component
- 4 - END

Please enter your choice (1, 2, 3 or 4) -> 3

Data item to be deleted: WASHINGTON

```
BOSTON  
CHICAGO  
DALLAS  
DENVER  
PITTSBURGH  
SAN FRANCISCO  
SEATTLE
```

Main menu:

- 1 - CREATE the linked list
- 2 - ADD a component
- 3 - DELETE a component
- 4 - END

Please enter your choice (1, 2, 3 or 4) -> 5

ERROR - Please try again

Main menu:

- 1 - CREATE the linked list
- 2 - ADD a component
- 3 - DELETE a component
- 4 - END

Please enter your choice (1, 2, 3 or 4) -> 4

End of computation

11.7 UNIONS

Unions, like structures, contain members whose individual data types may differ from one another. However, the members within a union all share the same storage area within the computer's memory, whereas each member within a structure is assigned its own unique storage area. Thus, unions are used to conserve memory. They are useful for applications involving multiple members, where values need not be assigned to all of the members at any one time.

Within a union, the bookkeeping required to store members whose data types are different (having different memory requirements) is handled automatically by the compiler. However, the user must keep track of what type of information is stored at any given time. An attempt to access the wrong type of information will produce meaningless results.

In general terms, the composition of a union may be defined as

```
union tag {
    member 1;
    member 2;
    . . .
    member m;
};
```

where **union** is a required keyword and the other terms have the same meaning as in a structure definition (see Sec. 11.1). Individual union variables can then be declared as

```
storage-class union tag variable 1, variable 2, . . . , variable n;
```

where **storage-class** is an optional storage class specifier, **union** is a required keyword, **tag** is the name that appeared in the union definition, and **variable 1**, **variable 2**, . . . , **variable n** are union variables of type **tag**.

The two declarations may be combined, just as we did with structures. Thus, we can write

```
storage-class union tag {
    member 1;
    member 2;
    . . .
    member m;
} variable 1, variable 2, . . . , variable n;
```

The **tag** is optional in this type of declaration.

EXAMPLE 11.33 A C program contains the following union declaration.

```
union id {
    char color[12];
    int size;
} shirt, blouse;
```

Here we have two union variables, **shirt** and **blouse**, of type **id**. Each variable can represent either a 12-character string (**color**) or an integer quantity (**size**) at any one time.

The 12-character string will require more storage area within the computer's memory than the integer quantity. Therefore, a block of memory large enough for the 12-character string will be allocated to each union variable. The compiler will automatically distinguish between the 12-character array and the integer quantity within the given block of memory, as required.

A union may be a member of a structure, and a structure may be a member of a union. Moreover, structures and unions may be freely mixed with arrays.

EXAMPLE 11.34 A C program contains the following declarations.

```
union id {
    char color[12];
    int size;
};
struct clothes {
    char manufacturer[20];
    float cost;
    union id description;
} shirt, blouse;
```

Now `shirt` and `blouse` are structure variables of type `clothes`. Each variable will contain the following members: a string (`manufacturer`), a floating-point quantity (`cost`), and a union (`description`). The union may represent either a string (`color`) or an integer quantity (`size`).

Another way to declare the structure variables `shirt` and `blouse` is to combine the preceding two declarations, as follows.

```
struct clothes {
    char manufacturer[20];
    float cost;
    union {
        char color[12];
        int size;
    } description;
} shirt, blouse;
```

This declaration is more concise, though perhaps less straightforward, than the original declarations.

An individual union member can be accessed in the same manner as an individual structure member, using the operators `.` and `->`. Thus, if `variable` is a union variable, then `variable.member` refers to a member of the union. Similarly, if `ptvar` is a pointer variable that points to a union, then `ptvar->member` refers to a member of that union.

EXAMPLE 11.35 Consider the simple C program shown below.

```
#include <stdio.h>

main()
{
    union id {
        char color;
        int size;
    };

    struct {
        char manufacturer[20];
        float cost;
        union id description;
    } shirt, blouse;

    printf("%d\n", sizeof(union id));

    /* assign a value to color */
    shirt.description.color = 'w';
    printf("%c %d\n", shirt.description.color, shirt.description.size);

    /* assign a value to size */
    shirt.description.size = 12;
    printf("%c %d\n", shirt.description.color, shirt.description.size);
}
```

This program contains declarations similar to those shown in Example 11.34. Notice, however, that the first member of the union is now a single character rather than the 12-character array shown in the previous example. This change is made to simplify the assignment of appropriate values to the union members.

Following the declarations and the initial `printf` statement, we see that the character 'w' is assigned to the union member `shirt.description.color`. Note that the other union member, `shirt.description.size`, will not have a meaningful value. The values of both union members are then displayed.

We then assign the value 12 to `shirt.description.size`, thus overwriting the single character previously assigned to `shirt.description.color`. The values of both union members are then displayed once more.

Execution of the program results in the following output.

```
2
w -24713
@ 12
```

The first line indicates that the union is allocated two bytes of memory, to accommodate an integer quantity. In line 2, the first data item (w) is meaningful, but the second (-24713) is not. In line 3, the first data item (@) is meaningless, but the second data item (12) has meaning. Thus, each line of output contains one meaningful value, in accordance with the assignment statement preceding each `printf` statement.

A union variable can be initialized provided its storage class is either `external` or `static`. Remember, however, that *only one member of a union can be assigned a value at any one time*. Most compilers will accept an initial value for only one union member, and they will assign this value to the first member within the union.

EXAMPLE 11.36 Shown below is a simple C program that includes the assignment of initial values to a structure variable.

```
#include <stdio.h>

main()
{
    union id {
        char color[12];
        int size;
    };

    struct clothes {
        char manufacturer[20];
        float cost;
        union id description;
    };

    static struct clothes shirt = {"American", 25.00, "white"};

    printf("%d\n", sizeof(union id));
    printf("%s %.2f ", shirt.manufacturer, shirt.cost);
    printf("%s %d\n", shirt.description.color, shirt.description.size);

    shirt.description.size = 12;
    printf("%s %.2f ", shirt.manufacturer, shirt.cost);
    printf("%s %d\n", shirt.description.color, shirt.description.size);
}
```

Notice that `shirt` is a static structure variable of type `clothes`. One of its members is `description`, which is a union of type `id`. This union consists of two members: a 12-character array and an integer quantity.

The structure variable declaration includes the assignment of the following initial values: "American" is assigned to the array member `shirt.manufacturer`; 25.00 is assigned to the integer member `shirt.cost`, and "white" is assigned to the union member `shirt.description.color`. Notice that the second union member within the structure, i.e., `shirt.description.size`, remains unspecified.

The program first displays the size of the memory block allocated to the union, and the value of each member of `shirt`. Then 12 is assigned to `shirt.description.size`, and the value of each member of `shirt` is again displayed.

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

```
12
American 25.00 white 26743
American 25.00 ~ 12
```

The first line indicates that 12 bytes of memory are allocated to the union, in order to accommodate the 12-character array. The second line shows the values initially assigned to `shirt.manufacturer`, `shirt.cost` and `shirt.description.color`. The value shown for `shirt.description.size` is meaningless. In the third line we see that `shirt.manufacturer` and `shirt.cost` are unchanged. Now, however, the reassignment of the union members causes `shirt.description.color` to have a meaningless value, but `shirt.description.size` shows the newly assigned value of 12.

In all other respects, unions are processed in the same manner, and with the same restrictions, as structures. Thus, individual union members can be processed as though they were ordinary variables of the same data type, and pointers to unions can be passed to or from functions (by reference). Moreover, most C compilers permit an entire union to be assigned to another, provided both unions have the same composition. These compilers also permit entire unions to be passed to or from functions (by value), in accordance with the ANSI standard.

EXAMPLE 11.37 Raising a Number to a Power This example is a bit contrived, though it does illustrate how a union can be used to pass information to a function. The problem is to raise a number to a power. Thus, we wish to evaluate the formula $y = x^n$, where x and y are floating-point values, and n can be either integer or floating point.

If n is an integer, then y can be evaluated by multiplying x by itself an appropriate number of times. For example, the quantity x^3 could be expressed in terms of the product $(x)(x)(x)$. On the other hand, if n is a floating-point value, we can write $\log y = n \log x$, or $y = e^{(n \log x)}$. In the latter case x must be a positive quantity, since we cannot take the log of zero or a negative quantity.

Now let us introduce the following declarations:

```
typedef union {
    float fexp;      /* floating-point exponent */
    int nexp;        /* integer exponent */
} nvals;

typedef struct {
    float x;          /* value to be raised to a power */
    char flag;        /* 'f' if exponent is floating-point,
                       /* 'i' if exponent is integer */
    nvals exp;        /* union containing exponent */
} values;

values a;
```

Thus, `nvals` is a user-defined union type, consisting of the floating-point member `fexp` and the integer member `nexp`. These two members represent the two possible types of exponents in the expression $y = x^n$. Similarly, `values` is a user-defined structure type, consisting of a floating-point member `x`, a character member `flag` and a union of type `nvals` called `exp`. Note that `flag` indicates the type of exponent currently represented by the union. If `flag` represents 'i', the union will represent an integer exponent (`nexp` will currently be assigned a value); and if `flag` represents 'f', the union will represent a floating-point exponent (`fexp` will currently be assigned a value). Finally, we see that `a` is a structure variable of type `values`.

With these declarations, it is easy to write a function that will evaluate the formula $y = x^n$, as follows.

```

float power(values a)      /* carry out the exponentiation */

{
    int i;
    float y = a.x;

    if (a.flag == 'i') {      /* integer exponent */
        if (a.exp.nexp == 0)
            y = 1.0;          /* zero exponent */
        else {
            for (i = 1; i < abs(a.exp.nexp); ++i)
                y *= a.x;
            if (a.exp.nexp < 0)
                y = 1./y;        /* negative integer exponent */
        }
    }
    else                      /* floating-point exponent */
        y = exp(a.exp.fexp * log(a.x));

    return(y);
}

```

This function accepts a structure variable (a) of type `values` as an argument. The method used to carry out the calculations depends on the value assigned to `a.flag`. If `a.flag` is assigned the character '`i`', then the exponentiation is carried out by multiplying `a.x` by itself an appropriate number of times. Otherwise, the exponentiation is carried out using the formula $y = e^{(n \log x)}$. Notice that the function contains corrections to accommodate a zero exponent ($y = 1.0$), and for a negative integer exponent.

Let us add a `main` function which prompts for the values of `x` and `n`, determines whether or not `n` is an integer (by comparing `n` with its truncated value), assigns appropriate values to `a.flag` and `a.exp`, calls `power`, and then writes out the result. We also include a provision for generating an error message if `n` is a floating-point exponent and the value of `x` is less than or equal to zero.

Here is the entire program.

```

/* program to raise a number to a power */

#include <stdio.h>
#include <math.h>

typedef union {
    float fexp;           /* floating-point exponent */
    int nexp;             /* integer exponent */
} nvals;

typedef struct {
    float x;              /* value to be raised to a power */
    char flag;             /* 'f' if exponent is floating-point,
                           'i' if exponent is integer */
    nvals exp;             /* union containing exponent */
} values;

float power(values a);    /* function prototype */

```

```

main()
{
    values a;                  /* structure containing x, flag and fexp/nexp */
    int i;
    float n, y;

    /* enter input data */
    printf("y = x^n\nnEnter a value for x: ");
    scanf("%f", &a.x);
    printf("Enter a value for n: ");
    scanf("%f", &n);

    /* determine type of exponent */
    i = (int) n;
    a.flag = (i == n) ? 'i' : 'f';
    if (a.flag == 'i')
        a.exp.nexp = i;
    else
        a.exp.fexp = n;

    /* raise x to the appropriate power and display the result */
    if (a.flag == 'f' && a.x <= 0.0)   {
        printf("\nERROR - Cannot raise a non-positive number to a ");
        printf("floating-point power");
    }
    else {
        y = power(a);
        printf("\ny = %.4f", y);
    }
}

float power(values a)      /* carry out the exponentiation */
{
    int i;
    float y = a.x;

    if (a.flag == 'i') {      /* integer exponent */
        if (a.exp.nexp == 0)
            y = 1.0;          /* zero exponent */
        else {
            for (i = 1; i < abs(a.exp.nexp); ++i)
                y *= a.x;
            if (a.exp.nexp < 0)
                y = 1./y;        /* negative integer exponent */
        }
    }
    else                      /* floating-point exponent */
        y = exp(a.exp.fexp * log(a.x));

    return(y);
}

```

Notice that the union and structure declarations are external to the program functions, but the structure variable **a** is defined locally within each function.

The program does not execute repetitiously. Several typical dialogs, each representing a separate program execution, are shown below. As usual, the user's responses are underlined.

```
Enter a value for x: 2
Enter a value for n: 3

y = 8.0000

Enter a value for x: -2
Enter a value for n: 3

y = -8.0000

Enter a value for x: 2.2
Enter a value for n: 3.3

y = 13.4895

Enter a value for x: -2.2
Enter a value for n: 3.3

ERROR - cannot raise a non-positive number to a floating-point power
```

It should be pointed out that most C compilers include the library function `pow`, which is used to raise a number to a power. We have used `pow` in several earlier programming examples (see Examples 5.2, 5.4, 6.21, 8.13 and 10.30). The present program is not meant to replace `pow`; it is presented only to illustrate the use of a union in a representative programming situation.

Review Questions

- 11.1 What is a structure? How does a structure differ from an array?
- 11.2 What is a structure member? What is the relationship between a structure member and a structure?
- 11.3 Describe the syntax for defining the composition of a structure. Can individual members be initialized within a structure type declaration?
- 11.4 How can structure variables be declared? How do structure variable declarations differ from structure type declarations?
- 11.5 What is a tag? Must a tag be included in a structure type definition? Must a tag be included in a structure variable declaration? Explain fully.
- 11.6 Can a structure variable be defined as a member of another structure? Can an array be included as a member of a structure? Can an array have structures as elements?
- 11.7 How are the members of a structure variable assigned initial values? What restrictions apply to the structure's storage class when initial values are assigned?
- 11.8 How is an array of structures initialized?
- 11.9 What is the scope of a member name? What does this imply with respect to the naming of members within different structures?
- 11.10 How is a structure member accessed? How can a structure member be processed?
- 11.11 What is the precedence of the period (.) operator? What is its associativity?
- 11.12 Can the period operator be used with an array of structures? Explain.
- 11.13 What is the only operation that can be applied to an entire structure in some older versions of C? How is this rule modified in newer versions that conform to the current ANSI standard?