Chapter 10

An Introduction to Arrays

The data types we have studied so far have been simple, predefined types and variables of these types are used to represent single data items. The real power of a computer language, however, comes from its ability to define complex, multipart, structured data types that model the complex properties of real-world objects. For example, to model the periodic table of the elements, we would need a collection of 110 objects that represent elements; each object would have several parts (name, symbol, atomic number, atomic weight, etc.). We call such types *compound types* or **aggregate types**. An array is an aggregate whose parts are all the same type. In this chapter, we study how to define, access, and manipulate the elements of an array. The last half of the chapter presents simple, important array algorithms.

10.1 Arrays

In many applications, each data item is processed once, just after it is read, and never needs to be used again. In such programs, an array can be used to store the data, but it is not necessary. In contrast, some programs must read all the data, perform a calculation, then go back and process the data again. In these programs, we must store all the data between reading it and reprocessing it.

Consider the problem of assigning grades to a class based on the average score of an exam. If we were computing only the exam average, this could be done without storing the data in an array; all that is needed is a summation loop. However, to assign a grade, we must have both the exam score and the average. The scores must be processed once to compute the average, then we must go back and reprocess the scores to assign the grades. In between we need to store the values.

We could do this using individual variables. For instance, in the example of computing the average of three numbers in Figure 2.7, we used three separate variables to hold the numbers. While this works well for only three numbers, it does not work on larger data sets. Imagine how tedious it would be to write a `scanf()` statement and a formula with many variable names. We can solve this problem by using an array. For example, a program to compute the average temperature over a 24-hour period might store

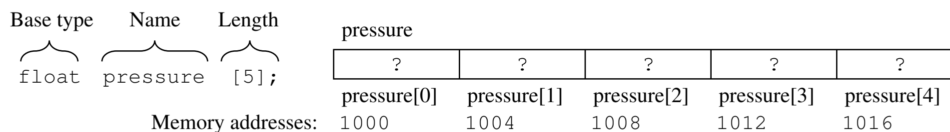


Figure 10.1. Declaring an array of five floats.

24 temperature readings in an array named `temperature`. An array used to determine the average high temperature for one year would have 365 (or 366) entries, each containing a daily high temperature.

An **array** is a consecutive series of variables that share one variable name. We will call these variables the **array slots**; the data values stored there are called the **array elements**. The number of slots in a *one-dimensional array* is called the **array length**. Arrays with two or more dimensions also can be defined; these will be studied in Chapter 18. The variables, or slots, that form an array have a uniform type called the **base type**.

An array object, as a whole, is given a name. We can refer to the entire array by this name or to an individual slot by appending a number in square brackets to the name. This number is called the *subscript*. A **subscript** is an integer expression enclosed in square brackets that, when written after an array name, designates a particular slot in the array.

In `C`, all arrays start with slot 0 (rather than 1) because it is easier and more efficient for the system to implement. This means that the first element in an array named `ary` would be called `ary[0]`; the next one would be `ary[1]`. If this array had six elements, the last one would be `ary[5]`.

10.1.1 Array Declarations and Initializers

Declarations. An array variable is declared and initialized very much like a simple variable. The **array declaration** starts with the base type, which can be any type—simple or compound. Thus, we can have an array of `int`s, an array of `char`s (also known as a *string*), or even an array of arrays. Following the base type in the declaration is the array name and a pair of square brackets enclosing the length, which must be an integer constant or an integer **constant expression** (an expression with only constant operands). The length determines the number of slots in the array. Figure 10.1 shows the declaration for an array named `pressure` containing five `float`s. This creates a series of five `float` variables that we can refer to as `pressure[0]`, `pressure[1]`, `pressure[2]`, `pressure[3]`, and `pressure[4]`. These five `float`s will be stored in a contiguous set of memory locations with `pressure[0]` at the location with the lowest memory address, 1000, in this case. In later chapters, the address of each slot may also be of interest; if so, we write the address above or below the slot, as shown in Figure 10.1.

To diagram an array, we draw a row of connected boxes that are the right size for the base type. We write the name of the array above and the subscripts below this row. If there is an initializer, as in Figure 10.2, we copy the initial values into the boxes. Otherwise, as in Figure 10.1, we leave them blank or write a question mark.

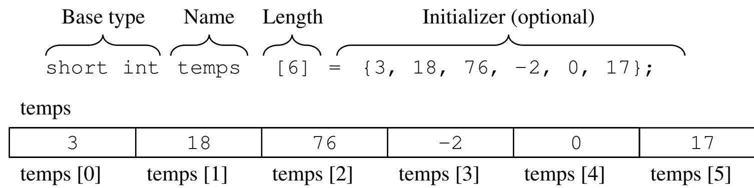


Figure 10.2. An initialized array of short `ints`.

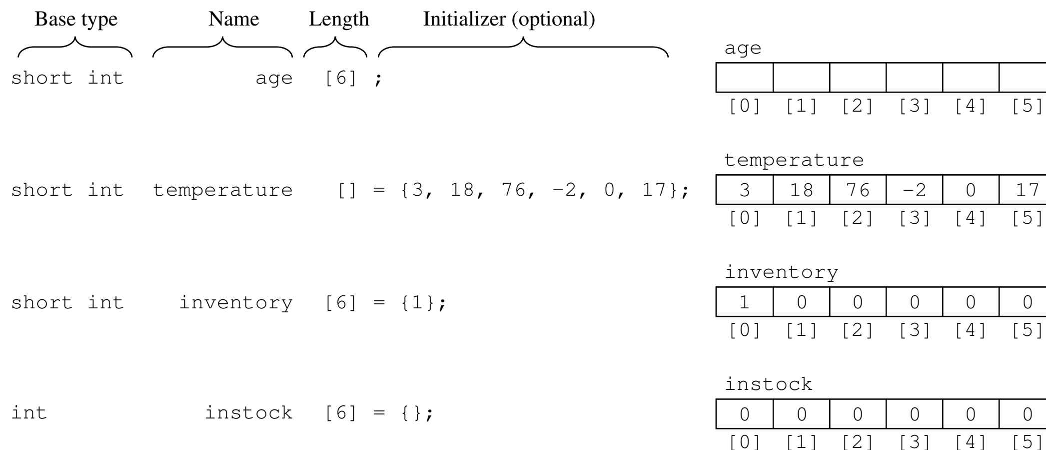


Figure 10.3. Length and initializer options.

Initial values. An array can be declared with no initial values, like the array named `pressure` in Figure 10.1. The contents of an uninitialized array are as unpredictable as ordinary variables.[1]

Alternatively, it may have an **array initializer**, which is a list of values enclosed in curly brackets. The values will be stored in the array slots when the array is created, as illustrated in Figure 10.2. The values in the initializer list must be constants or constant expressions. The types of values in the initializer must be appropriate for the base type of the array.[2]

If an initializer *is* given, the array length may be omitted from the square brackets. The `C` translator will count the number of initial values given and use that number as the length; exactly enough space will be allocated to store the given values. Note the absence of the length value in the declaration for `temperature` in Figure 10.3.

What if both a length and an initializer are given and the sizes do not match? This is an error if the

[1]Global arrays and static local arrays are initialized to 0 values.

[2]The initializer type must match or be coercible to the array base type.

```
#include <stdio.h>
#define DIMP 5    /* Lengths of the arrays. */
#define DIMT 6
#define DIMV 3
int main( void )
{
    float pressure[DIMP] = { .174, 23.72, 1.111, 721.2, 36.3 };
    short int temps[DIMT] = { 3, 18, 76, -2 };
    double vec2[DIMV] = { 2.0, 0.0, 1.0 };
    printf( " pressure: sizeof(float)  is %i * length %i ="
            " sizeof array %i \n", sizeof(float), DIMP, sizeof(pressure) );
    printf( " temps:    sizeof(short)  is %i * length %i ="
            " sizeof array %i \n", sizeof(short), DIMT, sizeof(temps) );
    printf( " vec2:     sizeof(double) is %i * length %i ="
            " sizeof array %i \n", sizeof(double), DIMV, sizeof(vec2) );
}
```

Figure 10.4. The size of an array.

initializer contains too many values; the compiler will detect this error and comment on it. However, if an initializer list is too short, it is not an error. In this case, the values provided are used for the first few array slots and the value 0 is used to initialize all remaining slots. The declaration for `inventory` on the third line in Figure 10.3 illustrates an initializer that is shorter than the array. It also is possible to initialize an entire array to zeros by supplying an empty set of brackets, as in the initializer for `instock` in Figure 10.3.

10.1.2 The Size of an Array

Two different aspects of an array's size are important: its length and the total number of bytes of memory required to store it. When we use `sizeof` with an array variable, we get the number of bytes, which is the product of the array's length and the size of one element of its base type.

While knowing the number of slots is necessary to write an array declaration, a programmer will not always know the **size of the array**, because that depends on the size of the base type, which may vary from one machine to another. The program in Figure 10.4 shows the sizes of the arrays declared in Figures 10.1, 10.2, and 10.5. An output from this program is

```
pressure: sizeof(float)  is 4 * length 5 = sizeof array 20
temps:    sizeof(short)  is 2 * length 6 = sizeof array 12
vec2:     sizeof(double) is 8 * length 3 = sizeof array 24
```

Often, the first portion of an array will hold data, while the last portion is not in use. This happens when the array is intended to hold a variable amount of information and its length is set to the maximum length that might ever be needed. Even in this case, the size returned by `sizeof` is the total amount of memory allocated including both the portion in use and the unused portion. This is illustrated by the array `temps` in Figure 10.4.

When an array is passed as an argument to a function, (see Section 10.4) the size information does not travel along with it. No operation in C will give us the actual length of an array argument inside a function. If you apply `sizeof` to an array parameter, the result always will be the number of bytes needed to store the starting address of the array. Unfortunately, an array-processing function frequently needs the information to work properly. For this reason, the programmer, who knows the array's length when it is declared, must make the information available for use by every part of the program that operates on the array. One way to do this is to declare the array length using a `#define` at the top of the program, as in our first several examples.

10.1.3 Accessing Arrays

The elements of an array can be accessed in two ways: by using pointers or by using subscripts. Both ways are important in C and need to be mastered. Pointers often are used when the slots of an array will be used in sequential numeric order, because this technique can lead to greater efficiency. While subscripts can be used for this purpose, too, they are better at accessing the elements in a random order. Since pointer processing techniques are harder to master than those based on subscripts, using pointers will be deferred until Chapter 17, while subscripting is explained in this chapter.

Subscripts. Figure 10.5 shows how constant subscripts are interpreted. It shows two arrays named `vec` and `vec2`, which represent vectors in a three-dimensional space. The first slot of `vec2` is `vec2[0]` and represents the x-component of the vector, containing the value 2.0. The y-component has the value 0.0 and is stored in `vec2[1]`, the shaded area in the diagram. In an object diagram, we use an arrow to represent an address. The arrow in Figure 10.5 represents `&vec2[1]`, the address of the shaded area. When both an ampersand and a subscript are used, the subscripting operation is done first and the "address of" operation is applied to the single variable selected by the subscript.[3] These addresses are important when using arrays as function arguments, as demonstrated in Section 10.4.

We also can use an expression involving a variable to compute a subscript (Figure 10.6). A subscript expression can be as simple as a constant or as complex as a function call, as long as the final value is a nonnegative integer less than the length of the array. The most frequently used subscript expression is a simple variable name, which also is illustrated in Figure 10.5. The phrase `vec2[k]` means that the current value of `k` should be used as a subscript for the array. Since `k` contains a 2 here, `vec2[k]` means `vec2[2]`, which contains the value 1.0. The flexibility and versatility of these subscript rules enables a variety of powerful array-processing techniques.

[3]Appendix B contains a precedence table that includes both subscript (`[]`) and address of (`&`) operators. Note that the precedence of subscript is higher, and therefore, the subscript will be applied to the array name before the address operator.

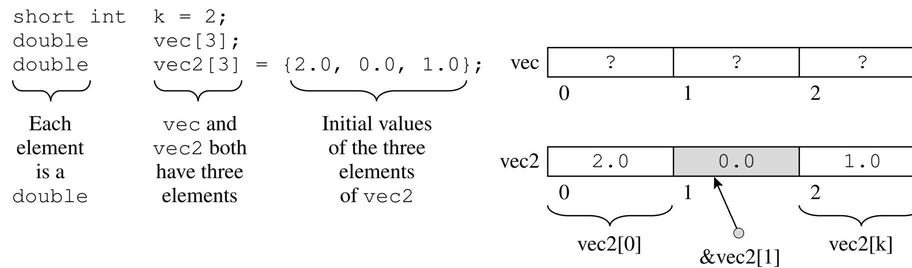


Figure 10.5. Simple subscripts.

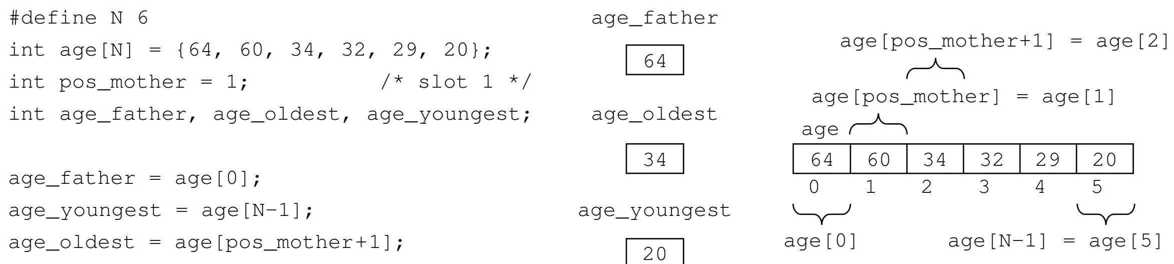


Figure 10.6. Computed subscripts.

Notes on Figure 10.6. Computed subscripts. Figure 10.6 demonstrates how we can use computed subscripts. It depicts an array containing the ages of N members of a family: father, mother, and children, in order of age. The array length is `#define`d. Often this is done so that it can be used for calculations throughout the code yet modified easily if the program's needs change.

The variable `pos_mother` is used here to store the position in the array of the mother. In the last line on the left, the expression `pos_mother+1` is used to find the age of the oldest child. Also, we often need to use the subscript of the last element in an array. To compute this, we subtract 1 from the array length. Therefore, `age[N-1]` is the age of the last (youngest) child in the family.

The example in Figure 10.7 declares an array variable named `vec` and shows simple input, computation, and output statements that operate on the array elements.

Notes on Figure 10.7. Subscript demo, the magnitude of a vector.

First box: input into an array. When we call `scanf()` to read the value of a variable, we write `&` before the variable's name to refer to its address. Similarly, we can use `&` to refer to the address of a single slot of an array. To read just one value into the first component, we would write `scanf( "%lg", &vec[0] );`.

This brief demonstration program shows how to do input, output, and computation with the elements of an array.

```
#include <stdio.h>

int main( void )
{
    double vec[3];      /* A vector in 3-space. */
    double magnitude;

    puts( "\n Subscript Demo: The Magnitude of a Vector" );

    printf( " Please enter the 3 components of a vector: " );
    scanf( "%lg%lg%lg", &vec[0], &vec[1], &vec[2] );

    magnitude = sqrt( vec[0] * vec[0] + vec[1] * vec[1] + vec[2] * vec[2] );

    printf( "    The magnitude of vector ( %g, %g, %g ) is %g \n",
            vec[0], vec[1], vec[2], magnitude );

    return 0;
}
```

Figure 10.7. Subscript demo, the magnitude of a vector.

Second box: computation on array elements.

- We can use a subscripted array name like a simple variable name in any expression.
- Here, to compute the magnitude of a vector, we add the squares of the three components, take the square root of the sum, and store the result in `magnitude`. The easiest and most efficient way to square a number is to multiply it by itself.

Third box: output from an array.

- To print an array element, give the array name and the subscript of the element.
- You cannot print the entire contents of an array with just one format field specifier. Here, we use three separate `%g` specifiers to print the three array elements. A loop would be used to print all the elements of a large array,
- The output from two runs of this program is

```
Subscript Demo: The Magnitude of a Vector
Please enter the 3 components of a vector: 1 0 1
   The magnitude of vector ( 1, 0, 1 ) is 1.41421
```

```
--------------------------------------------------

Subscript Demo: The Magnitude of a Vector
Please enter the 3 components of a vector: 1 2 0
   The magnitude of vector ( 1, 2, 0 ) is 2.23607
```

10.1.4 Subscript Out-of-Range Errors

One important reminder and caution: It is *up to the programmer* to ensure that all subscripts used are legal. C does not help you confine your processing to the array slots that you defined. C uses the subscript value to compute a memory address called the **effective address** according to this formula:

$$\texttt{effective_address} = \text{address of beginning of the array} + \text{subscript} \times \texttt{sizeof}\ (\text{base type of array})$$

If you use a subscript that is negative or too large, C will compute the theoretical "address" of the nonexistent slot and use that address even though it will not be between the beginning and the end of the array. C does absolutely no subscript range checking. The compiler will give no error comment and there will be no error comment at run-time either. Your program will run and either access a memory location that belongs to some other variable or attempt to access a location that does not exist.

10.2 Using Arrays

A common array processing pattern involves a counting loop, where the loop variable starts at 0 and stops before `N`, the length of the array. The loop variable is used as a subscript to access each array element, in turn. This kind of loop is seen again and again in programs that use arrays to process large amounts of data. Often, one loop is used to read data into the array, another to process the data items, and a third loop to print them all.

10.2.1 Array Input

The best way to read data into an array is with a `for` loop, where the loop counter starts at 0, increments through the subscripts of the array, and ends at `N`, the declared length of the array. The loop does not try to process slot `N`, which is past the end of an `N`-element array. This simple idiom is illustrated in Figure 10.8.

Notes on Figure 10.8. Filling an array with data.

Outer box: the* `for` *loop. Since the loop counter takes on the values from 0 to `N` and the loop ends when `k == N`, all `N` array slots (with subscripts `0...N-1`) will be filled with data.

Inner box: reading data into one slot. Within the loop, a `scanf()` statement reads one data value on each iteration and stores it in the next array slot. Note that both an ampersand and a subscript are used in the `scanf()` statement to get the address of the current slot.

```
#include <stdio.h>
#define N 3

int main( void )
{
    int k;                  /* Loop counter. */
    float dimension[N];     /* Dimensions of a box. */
    float volume;           /* The volume of the box. */

    printf( "\n Array Input Demo: the Volume of a Box \n"
            " Please enter dimensions of box in cm when prompted.\n" );

    for (k = 0; k < N; ++k) {  /* End loop when k reaches length of array. */
        printf( " > " );
        scanf( "%g", &dimension[k] );
    }

    volume = dimension[0] * dimension[1] * dimension[2] / 1e6;
    printf( " Volume of the box ( %g * %g * %g ) is %g cubic m.\n\n",
            dimension[0], dimension[1], dimension[2], volume );
    return 0;
}
```

Figure 10.8. Filling an array with data.

Sample output.

```
Array Input Demo: the Volume of a Box
Please enter dimensions of box in cm when prompted.
> 100
> 100
> 100
Volume of the box ( 100 * 100 * 100 ) is 1 cubic m.
```

10.2.2 Walking on Memory

A loop that runs amok can cause diverse kinds of trouble. One common outcome is that variables that occupy nearby memory locations are overwritten with information that was supposed to go into one of the array's slots. Afterward, any computation or output that uses these variables will be erroneous.

If you write a loop to print the values in an array and it loops too many times, you will start printing the values stored adjacent to the array in memory. When storing data, you will start erasing other information when the loop exceeds the array bounds. This is demonstrated by the program in Figure 10.9 (a modification

This program is a modification of the vector magnitude program in Figure 10.7. It demonstrates how an incorrect loop can destroy the value of a variable and result in unpredictable behavior.

```
#include <stdio.h>

int main( void )
{
    int k;              /* Loop counter. */
    float v[3];         /* A vector in 3-space. */
    float sum;          /* The sum of the squares of the components. */
    float magnitude;

    puts( "\n Subscript Demo: Walking on Memory" );
    printf( " Please enter one float vector component at each prompt.\n" );

    for (sum = 0.0, k = 0; k <= 3; ++k) {       /* Loop goes too far. */
        printf( "\tv[%i]: ", k );
        scanf( "%g", &v[k] );
        sum += v[k] * v[k];
    }
    magnitude = sqrt( sum );
    printf( "    The magnitude of vector ( %g, %g, %g ) is %g \n",
            v[0], v[1], v[2], magnitude );
    return 0;
}
```

Figure 10.9. Walking on memory.

of the program in Figure 10.7), which reads input into an array named `v`. Figure 10.10 is a diagram of memory for this program. It shows the variables and the memory addresses that a typical compiler might assign. The array is colored gray.

The array has three slots, but the loop was written to execute four times, with subscripts 0 through 3. The last time, the effective address calculated for `v[k]` actually is the address of `k`, and the input value, wrongly, is stored on top of the loop counter. The output from this run had more lines than expected:

```
Subscript Demo: Walking on Memory
Please enter one float vector component at each prompt.
       v[0]: 0.0
       v[1]: 1.0
       v[2]: 2.0
       v[3]: 0.0
       v[1]: 5.0
       v[2]: -1.0
       v[3]: 4.5
Segmentation fault
```

The upper diagram in Figure 10.10 shows the values of the variables just after the third trip around the

These diagrams illustrate the contents of memory while processing the input sequence described in the text.

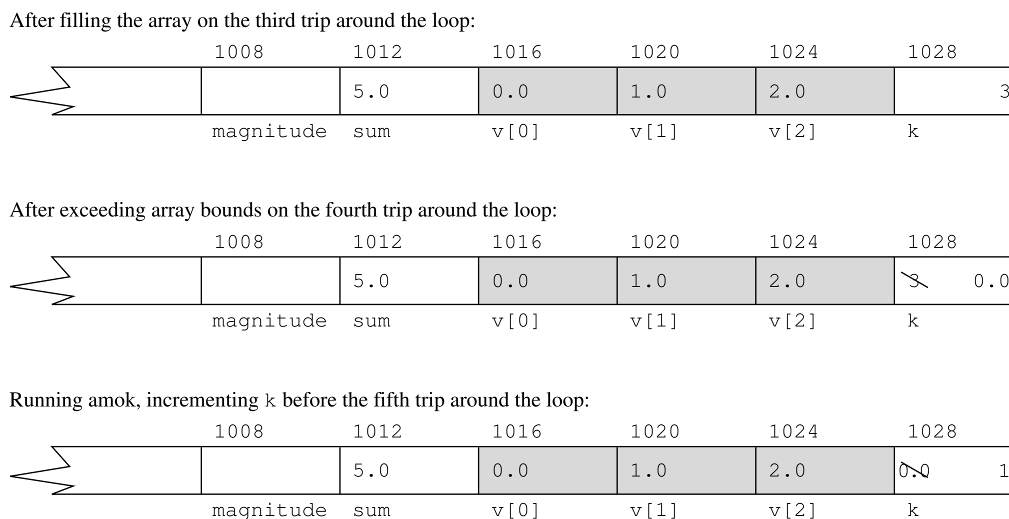


Figure 10.10. Before and after walking on memory.

loop. All the array slots have been filled, and `k` has been incremented to 3 and is ready for the loop exit test. However, since the test was written incorrectly (with a `<=` operator instead of a `<` operator), the loop does not end.

The middle diagram shows what happens during the next loop iteration, as the subscript goes beyond the end of the array. The input value of 0.0 is stored in the variable that follows the array, which is the memory location used for the loop counter. This destroys the value of the loop counter and leaves the input value in its place. In this example, the input is 0, which then gets incremented to 1 by the `for` loop (bottom diagram). The loop still does not terminate, because now `k` is 1. It continues taking input until some input value is stored in `k` that satisfies the loop exit test or until an abnormal termination happens, as occurs here.

In this example, storing input on top of the loop counter causes unpredictable behavior that depends on the data entered by the user. Usually, as in the output shown, the program crashes; other times it continues and terminates normally but produces erroneous results. Be sure to check the limits of array processing loops to minimize these problems.

Random locations and memory faults. Sometimes, a faulty subscript causes the program to try to use a memory location that is not legal for that program to access; the result is an immediate hardware error (a memory fault, bus error, or segmentation fault) that terminates the program. Usually, the system displays a message to this effect, as was seen in the last program.

Problem scope: Given the ID number and two exam scores (midterm and final) for each student in a class, compute the weighted average of the two scores. Also, compute the overall class average and the difference between that and each student's average.

Input: ID numbers will be long integers and exam scores will be integers.

Limitations: The class cannot have more than 16 students.

Formula: Weighted average = $0.45 \times$ midterm score + $0.55 \times$ final score

Output and computational requirements: All inputs should be echoed. In addition, for each student, print the exam average and the difference between that average and the overall average for the class, both to one decimal place. The overall exam average of the class should be printed using two decimal places.

Figure 10.11. Problem specifications: Exam averages.

If a program continues to run after a subscript error, it generally produces erroneous output. The cause of the errors may be difficult to detect because the value of some variable can be changed by a part of the program that does not refer to that variable at all, and output based on the mistake may not occur until long after the destructive deed. Prevention is the best strategy for developing working code. It is up to the programmer to use subscripts carefully and ensure that every subscript is legal for its array. With this in mind, remember that

- Arrays start with subscript 0, so the largest legal subscript is one less than the number of elements in the array.
- An input value must be checked before it can be used as a subscript. Negative values and values equal to or larger than the array length must be eliminated.
- A counting loop that processes an array should terminate when the loop counter reaches the number of items in the array.

10.3 Parallel Arrays

The next program (specified in Figure 10.11 and given in Figure 10.12) uses a set of **parallel arrays**, all of the same length, to implement a table of data. Each array in the set represents one column of the table and each array subscript represents one row of data. In this example, the first column is a list of student ID numbers. Parallel to it are three other arrays containing data about the students. The data at subscript `k` in each of these arrays corresponds to the student with subscript `k` in the ID array. This is illustrated by the declarations and diagram in Figure 10.13.

When a table is implemented as a set of parallel arrays, the same variable is used to subscript all of them. We can apply this principle here. A loop is used to select the array slots. For each slot, first, input is read into three of the arrays at the selected position, then an average is calculated and stored in the fourth array. After the input loop, we scan this last array to compute several more values.

Notes on Figure 10.12. Using parallel arrays.

```
#include <stdio.h>
#define MAX 16
int main( void )
{   int k;              /* Loop counter. */
    int n;              /* Number of students in class. */
    long id[MAX];       /* Students' ID numbers. */
    short midterm[MAX], final[MAX];     /* Exam scores. */
    float average[MAX]; /* Average of exam scores. */
    float avg_average;  /* Average of averages. */
    float diff;         /* Student's average minus class average. */
    printf( " Exam average = .45*midterm + .55*final.\n"
            " How many students are in the class? " );
    scanf( "%i", &n );
    if (n > MAX || n < 1) {
            printf( "Size must be between 1 and %i.", MAX );
            exit(1);
    }

    printf( " At each prompt, enter an ID# and two exam scores.\n" );
    for (k = 0; k < n; ++k) {
        printf( "\t > " );
        scanf( "%li%hi%hi", &id[k], &midterm[k], &final[k] );
        average[k] = .45 * midterm[k] + .55 * final[k];
    }

    for (avg_average = 0, k = 0; k < n; ++k) avg_average += average[k];
    avg_average /= n;
    printf( "\nAverage of the averages = %.2f\n", avg_average );

    puts( "\nID num   mid   fin   average   +/- " );
    puts( "----------------------------------- " );

     for (k = 0; k < n; ++k) {
         diff = average[k] - avg_average;
         printf( "%li %5hi %5hi %8.1f %6.1f \n",
                 id[k], midterm[k], final[k], average[k], diff );
     }

    puts( "-----------------------------------" );

    return 0;
}
```

Figure 10.12. Using parallel arrays.

This is a diagram of the memory for the program in Figure 10.12. A set of parallel arrays is used to represent the exam scores and exam average for a class. A common subscript, `k`, is used to subscript all four arrays. The maximum number of students this table can hold is 16, but this class has only 13 students, so the last three array slots are empty.

```
#define MAX 16

int k;
int n;
long id[MAX];
short midterm[MAX];
short final[MAX];
float average[MAX];
```

MAX:16

k: 3

n: 13

	id	midterm	final	average
0	825176	80	85	82.8
1	825301	72	68	69.8
2	824769	97	90	93.2
3	826162	57	66	62.0
4	824388	88	92	90.2
5	825564	42	61	52.5
6	825923	75	62	67.8
7	823976	82	81	81.4
8	824662	91	94	92.7
9	824478	68	80	74.6
10	826056	82	71	75.9
11	826178	95	97	96.1
12	825743	51	57	54.3
13				
14				
15				

Figure 10.13. Parallel arrays can represent a table.

First box: limiting the subscripts.

- Serious errors result from using a subscript beyond the end of the array. To avoid this, we check that the class size is within the limits we are prepared to handle. If it is too large, we abort. We also abort if the size is negative or 0, because these values are meaningless.
- If a class really had more than 16 students, this program would need to be edited to make `MAX` larger and then recompiled, so we print an error comment and end execution gracefully.

Second box: the input phase.

- Our input loop counts from 0 up to the class size the user has entered. Since this count has been validated, we can be sure that all array subscripts are legal.
- On each iteration we enter all the data for one student. Three numbers are read and stored in the corresponding slots of the first three arrays.

- Sometimes the input loop does only input and other loops are used to process the data. In this example, the loop both reads the input and calculates the weighted average, which is part of a student's record and based directly on the input. This average is stored in the fourth array (the fourth column of the table). Merging these actions leads to a slightly more efficient program. However, if the additional calculations are lengthy, efficiency can be sacrificed for the added clarity of splitting the tasks into separate loops.
- The prompts and input process look like this:

```
Exam average = .45*midterm + .55*final.
How many students are in the class? 13
At each prompt, enter an ID# and two exam scores.
        > 825176 80 85
        > 825301 72 68
        > 824769 97 90
        > 826162 57 66
        > 824388 88 92
        > 825564 42 61
        > 825923 75 62
        > 823976 82 81
        > 824662 91 94
        > 824478 68 80
        > 826056 82 71
        > 826178 95 97
        > 825743 51 57
```

- We will echo the input data later, along with calculated values.

Third box: the average calculation.

- We sum the weighted averages as the first step of computing the overall class average. This task also could have been done as part of the input loop but was written as a separate loop because it has no direct connection to the input process or a single student's record.
- We use a one-line `for` loop, because summing the values in an array is a simple job that corresponds to a single conceptual action.
- Note how convenient the `+=` operator is for summing the elements of an array. The `/=` operator provides a concise way to say "now divide the sum by the number of students."
- The output from this box is

```
Average of the averages = 76.40
```

Fourth and fifth boxes: the output phase.

- In the outer box, we print table headings before the output loop and print a line to terminate the table after the loop.
- In the inner box, we print each student's record by including one value from each of the parallel arrays and a final value computed from the average array. Note that we use `%f` in the `printf()` format to make nicely aligned columns.
- The final output of the program is

```
ID num   mid   fin   average   +/-
------------------------------------
825176    80    85      82.8    6.3
825301    72    68      69.8   -6.6
824769    97    90      93.2   16.8
```

```
826162    57    66     62.0  -14.5
824388    88    92     90.2   13.8
825564    42    61     52.5  -24.0
825923    75    62     67.8   -8.6
823976    82    81     81.4    5.0
824662    91    94     92.7   16.2
824478    68    80     74.6   -1.8
826056    82    71     75.9   -0.5
826178    95    97     96.1   19.7
825743    51    57     54.3  -22.1
---------------------------------
```

- The input data is printed side by side with the final output to make it easier to check whether the computations are correct.

10.4 Array Arguments and Parameters

No data type is very useful in a programming language unless it can be used in a function call to pass information into and out of a function. Therefore, we need to know how to write a function with an array parameter and how to call such a function with an array argument. C does not permit a function to *return* an array value.[4]

Array arguments in C are handled differently from other types of arguments. When an `int` , a `double`, or a single element from an array is passed to a function, its value is copied into the parameter variable that has been created for the function. Technically, we say that arguments of simple types are passed *by value*. However, when an array is passed to a function, it is passed *by reference*, that is, only the *address* of the first slot of the array, not its entire list of values, is copied into the function's memory area. This is similar to the way that `scanf()` works. The address of a variable is passed to `scanf()`, which fills it with information from the keyboard, and this information remains in the variable even after `scanf()` finishes.

Passing array arguments by reference permits a large amount of data to be made available to a function efficiently (since the actual data are not copied) and also allows the function to store information into the array. Therefore, a program can pass an empty array into a function, which then will fill it with information. When the function returns, that information still is in the original array's memory and can be used by the caller.

To call a function with an **array argument**, the caller simply writes the name of the array and does *not* write a subscript or the square subscript brackets. Also, no `&` operator is used in front of the array name, because the array name automatically is translated into its starting address. To declare the corresponding **array parameter**, however, we use empty square brackets (with no length value). The length may be written between the brackets but it will be ignored by the compiler. This is done in C so the function can be used with arrays of many different lengths.

For example, if the actual argument were an array of `double`s, a formal parameter named `ara` would be declared as `double ara[]`. Within the function, the parameter name is used with subscripts to address the corresponding argument values.

[4]However, it is possible to return a pointer to an array. This topic is deferred until a later chapter.

The next program illustrates the basic array operations described so far: input, output, access, calculation, and the use of an array parameter. In it, the term `FName` means function name and `AName` means array name. We introduce and demonstrate the use of three new forms of function prototypes that manipulate arrays:

- `void FName( double AName[], int n );`

  A prototype of this form is used when the purpose of the function is to read data into the array or print the array data. The `get_data()` function in the next example has this form; its prototype is

  ```
  void get_data( double x[], int n );
  ```

  This function takes two parameters, an array of `double`s called `x` and an integer that gives the length of the array. Since the declaration of the parameter `x` contains no length, we need a limiting value. This can be the globally defined constant that was used to declare the array object. Often, though, we do not use the entire array, and a parameter is used to communicate the amount of the array currently in use. The `get_data()` function fills the array with input values that remain in it after the function returns. This is one way of returning a large number of values from a function to the calling program. Since there is no other return value, the return type is declared as `void`.

- `double FName( double AName[], int n );`

  A prototype of this form is used when an array contains data and we wish to access those data to calculate some result, which then is returned. The function named `average()` in the next example has this form; its prototype is

  ```
  double average( double x[], int n );
  ```

  It again takes two parameters, the array of `double`s and the current length of the array. The function calculates the average (mean) of those values and returns it to the caller via the `return` statement. Therefore, the function return type is declared as `double`.

- `double FName( double AName[], int n, double Arg );`

  We use a prototype of this form when we need both an array of data and another data value to perform a calculation. The function named `divisible()` in the next example has this form; its prototype is

  ```
  int divisible( int primes[], int n, int candidate );
  ```

  It takes three parameters, an array of prime numbers, its length `n`, and the `candidate` number we wish to test for primality. The numbers in the array are used to test the candidate; the answer will be *true* (1) or *false* (0).

10.5 An Array Application: Prime Numbers

The next application is a prime number generator that illustrates the use of arrays and array parameters. The task specifications are given in Figure 10.14, the main program in Figure 10.15, and a function in Figure 10.16.

Problem scope: Print a list of all prime numbers, starting with 2, and continuing until `MANY` primes have been printed. A prime number is an integer that has no proper divisors except itself and 1.

Constant: `MANY`, the number of primes to be found and printed.

Restrictions: `MANY` must be small enough that an array of `MANY` integers can fit into memory and the last prime calculated is less than the maximum integer that can be represented.

Input: None.

Output required: A neat list of primes, one per line.

Figure 10.14. Problem specifications: A table of prime numbers.

Notes on Figure 10.15. Calculating prime numbers.

First box: prototype.

- This prototype follows the third pattern discussed above: the parameters are an array, its length, and another value that must be used with the array.
- The `divisible()` function compares the `candidate` number to the numbers in the array. If the `candidate` is divisible by anything in the array, *true* (1) is returned. Otherwise it is non divisible (prime), so *false* (0) is returned.

Second box: the table of primes.

- The table will be filled in with prime numbers. The list will be generated in order by testing every possible odd number, starting with 3. As primes are discovered, we store them in the table. To test each integer, we use the previously computed portion of the table.
- We choose an arbitrary constant for the length of this table. Computing more primes requires more time and storage space. This method is limited by the space available and the largest integer that can be represented in the ordinary way. The latter limit usually occurs first.
- The first prime, and the only even prime, is 2. We initialize the first slot in the table to 2 so that the computation loop can be limited to testing odd numbers. The rest of the prime table will be initialized to 0.
- We already have stored one prime in the table, so we initialize `n` to 1. It will be incremented each time a new prime is found.

Third box: filling the table.

- We use a `for` loop that starts at 3 and counts by twos to test all the odd numbers.
- This is a very unusual loop. While we initialize `k` and increment it each time around the loop, we use `n`, the number of primes, to end the loop. We want to continue searching for primes until the table is filled;

This main program calls the functions in Figure 10.16. It calculates and prints a table of the first MANY prime numbers. Strategy: identify prime numbers in ascending order and print them. Save each prime in a table and use them all to test the next number in sequence.

```
#include <stdio.h>
#define MANY 3000

void print_table( int primes[], int num_primes );
int divisible( int primes[], int n, int candidate );   /* Is it nonprime? */

int main( void )
{
    int k;                  /* Integer being tested. */
    int primes[MANY]={2};  /* To begin, put the only even prime in table. */
    int n = 1;             /* Number of primes currently in table. */

    printf( "\nA Table of the First %i Prime Numbers\n", MANY );
    for (k = 3; n <= MANY; k += 2) {       /* Test the next odd integer.  */
                                           /* Quit when table is full.  */
        if (!divisible( k, primes, n )) {  /* If it is a prime... */
            primes[n] = k;                 /* ... put it in the table... */
            ++n;                           /* ... and count it. */
        }
    }
    print_table( primes, MANY );           /* Print table of primes. */
    return 0;
}
```

Figure 10.15. Calculating prime numbers.

that is, `n==MANY`. Since we do not know how big `k` will be at that time, we do not use `k` to terminate the loop.

Inner box: calling the function to test for primality.

- By definition, N is prime if it has no divisors except itself and 1. If N did have a divisor, it would have to have two, and one of them would have to be less than or equal to $\sqrt{N}$. Also, if N did have a divisor, D, either D would be a prime number or D itself would have at least two other divisors smaller than itself. Thus, we can show that N is a prime by showing that it is not divisible by any prime less than or equal to $\sqrt{N}$.
- If the `divisible()` function returns 1 (true), `k` is not a prime. If it returns 0 (false), we put `k` into the

table and increment `n`.

Last box: printing the table. The output is printed by calling `print_table()`. The first and last portions of it are

```
A Table of the First 3000 Prime Numbers
         2
         3
         5
         7
         9
        11
        13
        15
        17
        19
    ......
      5993
      5995
      5997
      5999
 ---------------------
```

Notes on Figure 10.16. Functions for the prime number program. The `divisible()` function can identify primes up to the square of the largest prime currently stored in the table. Its parameters are `candidate` (a number to test), `primes` (a table of primes), and `n` (the current length of the table).

First and third boxes: the termination condition.

- We define a variable of type `int`, whose value will be returned later as the result of the function. We initialize the variable to 0 (false) and later set it to 1 (true) if we find what we are searching for; that is, a number that evenly divides the candidate.
- We return the value of `found` after the search either succeeds or exhausts the data in the table.
- This is a common control pattern and especially useful when several tests must be made and any one of them could terminate processing.

Second box: the search loop.

- We use the modulus operator to test whether one number is divisible by another; a is divisible by b if the remainder of `a/b` is 0; that is, if `a%b == 0`.
- To test a candidate number, we divide it by all the numbers in the table up to the square root of the candidate and leave the search loop with a `break` statement the first time we find a proper divisor.
- If no divisor is found, one of two conditions will terminate the loop: Either we have tested every prime in the table or the next prime in the table is greater than the square root of the candidate.

These functions are called from Figure 10.15.

```
/* ---------------------------------------------------------------------
// Test candidate number for divisibility by primes in the table.
// Return true if a proper divisor is found, false otherwise.
*/
int
divisible( int primes[], int n, int candidate ) {
    int m;              /* Loop counter */
    int last = (int) sqrt( candidate );
    int found = 0;        /* Initially false, no divisor has been found. */

    /* Divide by every prime < square root of candidate. */
    for (m = 0; m < n && primes[m] <= last; ++m) {
        if (candidate % primes[m] == 0) {
            found = 1;    /* Set to true; divisor has been found. */
            break;
        }
    }

    return found;
}
/* ---------------------------------------------------------------------
// Print the list of prime numbers, one per line.
*/
void print_table( int primes[], int num_primes )
{
    int m;                /* Loop index for primes table */
    for (m = 0; m < num_primes; m++) printf( "%10i\n", primes[m] );
    printf( " --------------------\n" );
}
```

Figure 10.16. Functions for the prime number program.

10.6 Searching an Array

A common application of arrays is to store a table of data that will be searched, and possibly updated repeatedly, in response to user inputs. In the noncomputer world, a table has at least two columns: a column of index values and one or more columns of data. For example, in a periodic table of the elements, the atomic numbers (1...109) are used as the **index column**, then the element names, atomic weights and chemical symbols are **data columns**. If we use the same data for other purposes, a different column, such as the name, might be chosen as the index column.

A table can either be sorted or unsorted. The data in a **sorted table** are arranged in ascending or descending order, according to some comparison function defined on the values in the index column. For example, a periodic table is sorted in ascending order by the atomic number. A dictionary is sorted in ascending alphabetic order. The typical university course catalog is sorted in ascending order by department code, and within a department, by course number.

To implement a table in the computer, we can use either a set of parallel arrays or an **array of structures**[5]. When we implement a table as a set of **parallel arrays**, we use one array to represent the index column and one more for each data column in the table.

As discussed in Chapter 6, a typical **search loop** examines a set of possibilities, looking for one that matches a given key value. A sequential search of a table examines the index column for an entry that matches the key, one item after another. In every table-searching application, we find the following elements:

- A table, consisting of an index column and one or more data columns.
- A **search key**, the input value that must be compared to the entries in the index column of the table.
- A **comparison function** that is defined for the base type of the array. With simple types, such as numbers or characters, the `==` operator is appropriate. However, a programmer must define some other comparison function to search an array whose base type is an aggregate type such as those defined in Chapters 12 and 13.
- The **position variable**, the output from the search process, set to the subscript that identifies the value in the index column matching the key value.
- A **success or failure code**, sometimes a separate output value, other times failure may be indicated by setting the position variable to a value either too large or too small to be a legal subscript.

The next program example shows a search loop used for a simple application: recording bill payments in an array of account information. Figure 10.17, gives the specification, Figure 10.18 is the main program, and Figure 10.20 is the **sequential search** function implemented by a search loop.[6]

Notes on Figure 10.18. Main program for sequential search.

[5] Structures are explained in Chapter 13. If a table is modeled as an array of structures, the structure has one member for each column in the table. The array of structures usually is considered a better style because it is more coherent; that is, it groups together all the values for a table entry. The relative merits of these two approaches are discussed in Chapter 13.

[6] Discussion of the binary search algorithm, which is more complex but much faster for sorted arrays, is deferred until recursion is introduced in Chapter 19.

Problem scope: Starting with a list of payments that are due, record payment amounts and print a list of account balances after recording the payments.

Inputs: Input will happen in two phases.
Phase 1. For each account, enter the account ID number and the initial amount due.
Phase 2. The payments will be entered. For each one, the ID number is entered first. If it is found in the list of accounts, the user will be prompted for a payment amount.

Constants: `ACCOUNTS` must be defined as the maximum number of accounts that the company has simultaneously. The actual number of accounts can be smaller than this limit.

Output: For each account number entered in input phase 2, the position of that account in the list will be displayed. At the end of Phase 2, a list of final balances will be displayed. If the bill was overpaid, this balance will be negative.

Formulas: Each payment amount should be subtracted from the initial balance in the account.

Limitations: No attempt is made to validate payment amounts. ID numbers that are not in the list of accounts will cause an error comment but otherwise have no effect.

Figure 10.17. Problem specifications: Recording bill payments.

First box: prototypes for this application. The main program will call these three functions to do all the work. The first two are more or less the same in every array application that works on parallel arrays: an input function that fills the arrays and an output function that prints them.

Second box: modeling a table. We use a parallel-array data structure to implement a table. It has an index column (`ID`) and one data column (`owes`). The maximum capacity of the table is defined at the top of the program, (20 in this case) and the actual number of rows in the table will be determined at run time and stored in `n`.

Third box: variables for the search. A sequential search function tries to locate a key value in an array. If it is located, `where` will be used to store its position.

Fourth box and last box: input, echo, and final output. We call functions to read the input. In a realistic program, the data would be input from a file rather than from the keyboard. With only minor changes, the code in `get_data` can be changed to read the data from a file.

The output function is called twice: once to echo the input and again to print the results. When the output code is written in a function, it is easy to use it more than once. This can be especially useful during debugging, when one might wish to see the data in the array within the loop after every change.

```
#include <stdio.h>
#define ACCOUNTS 20

int  get_data( int ID[], float owes[], int nmax );
void print_data( int ID[], float owes[], int n );
int  sequential_search ( int ID[], int n, int key );

int main( void )
{
    int n;                  /* #of ID items; will be <=ACCOUNTS.     */
    int ID[ ACCOUNTS ];     /* ID's of members with overdue bills.   */
    float owes[ ACCOUNTS ]; /* Amount due for each member.           */

    int key;                /* ID number to search for.              */
    int where;              /* Position in which key was found.      */

    float payment;           /* Input: amount of payment for one ID#. */

    n = get_data( ID, owes, ACCOUNTS );   /* Input all unpaid bills. */
    printf( "\nInitial List of Unpaid Bills:\n     ID  Amount\n" );
    print_data( ID, owes, n );            /* Echo the input */

    printf( "\n%i items were read; ready for payments.\n\n", n );
    for (;;) {
        printf( "Enter ID number (zero to end): " );
        scanf( "%i", &key );
        if (key==0) break;

        where = sequential_search( ID, n, key );
        if (where<0) printf( " Item %i \t not found.", key);
        else {
            printf( "\tID %2i \t found in slot %i. ", key, where );
            printf( " Amount paid: " );
            scanf( "%g", &payment );
            owes[where] -= payment;
        }
    }
    printf( "\n\nFinal Amounts Due:\n     ID  Amount\n" );
    print_data( ID, owes, n );      /* Print final amounts owed. */
    return 0;
}
```

Figure 10.18. Main program for sequential search.

Sample output up to this stage. The first block of output came from `get_data()`, the second block from `print_data()`.

```
Enter pairs of ID # and unpaid bill (two zeros to end):
31        2.35
7         3.19
6         2.28
13        1.09
22        8.83
38        13.25
19        5.44
32        6.90
25        1.70
3         41.
0         0

Initial List of Unpaid Bills:
     ID  Amount
[ 0] 31    2.35
[ 1]  7    3.19
[ 2]  6    2.28
[ 3] 13    1.09
[ 4] 22    8.83
[ 5] 38   13.25
[ 6] 19    5.44
[ 7] 32    6.90
[ 8] 25    1.70
[ 9]  3   41.00

10 items were read; ready for payments.
```

Fifth box: payments and account balances. We have read an ID number and are ready to process a payment from that person. First, we must find the position of the person in the table; the call on `sequential_search()` does this, and stores the answer in `where`.

If the ID is not found in the table, `where` will have a negative value. Otherwise, its value will be between 0 and `n-1`. We check this condition, and go on to input and process a payment if the search was successful. Because this is a parallel-array data structure, the payment amount is subtracted from the bill at position `where` in the `owes` array which corresponds to the person at position `where` in the `ID` array.

Sample output from this phase.

```
Enter ID number (zero to end): 31
ID 31   found in slot 0.  Amount paid: 2.35
Enter ID number (zero to end): 25
ID 25   found in slot 9.  Amount paid: 2
Enter ID number (zero to end): 13
ID 13   found in slot 3.  Amount paid: 1
Enter ID number (zero to end): 38
ID 38   found in slot 5.  Amount paid: 10
Enter ID number (zero to end): 0

Final Amounts Due:
     ID  Amount
[ 0] 31    0.00
[ 1]  7    3.19
[ 2]  6    2.28
```

```
/* ---------------------------------------------------------------------- */
int                                    /* Actual number of data sets read. */
get_data( int ID[], float owes[], int nmax )
{
    int k;                             /* Loop counter and array index */

   printf( "Enter pairs of ID # and unpaid bill (two zeros to end):\n" );
   for (k=0; k<nmax; ++k) {           /* Don't go beyond end of arrays. */
       scanf( "%i%g", &ID[k], &owes[k] );
       if( ID[k]==0 ) break;          /* No more data is available. */
   }
   return k;
}
/* ---------------------------------------------------------------------- */
void print_data( int ID[], float owes[], int n )
{
    int k;                         /* Loop counter and array index */
    for (k=0; k<n; ++k) {          /* Don't read beyond end of ID array.*/
        printf( "[%2i] %2i %7.2f{\bk}n", k, ID[k], owes[k] );
    }
}
```

Figure 10.19. Input and output functions for parallel arrays.

```
[ 3] 13    0.09
[ 4] 22    8.83
[ 5] 38    3.25
[ 6] 19    5.44
[ 7] 32    6.90
[ 8]  3   41.00
[ 9] 25   -0.30
```

Termination. Both the `get_data()` function and the payment-processing loop terminate when a sentinel value (an ID number of 0) is entered.

Notes on Figure 10.19. Input and output functions for parallel arrays.

Sequential array processing. These two functions and the one in Figure 10.20 follow a common pattern used for processing an array sequentially. Each function uses a `for` loop to perform an operation (I/O or calculation) on every array element, starting with the first and ending with the last. A programmer can use arrays for a wide variety of applications by following this pattern and varying the operation.

The `get_data()` ***function.***

- We use a `for` loop to process the array. Before entering the loop, we prompt the user to enter a series of data values. Within the loop, we use a very short prompt to ask the user individually for each value.

This is a clear and convenient interactive user interface. During execution of `get_data()`, the user will see something like this:

```
Please enter data values when prompted.
x[0] = 77.89
x[1] = 76.55
x[2] = 76.32
x[3] = 79.43
x[4] = 75.12
x[5] = 64.78
x[6] = 79.06
x[7] = 78.58
x[8] = 75.49
x[9] = 74.78
```

- In the future, when we read data from a file, an interactive prompt will not be necessary.
- The variable `k` is used as the counter for the loop and also to subscript the array (the usual paradigm for processing arrays). We initialize `k` to 0 and leave the loop when `k` exceeds the last valid subscript, based on the current number of array slots that are in use.
- We also leave the loop if the user enters the sentinel signal: a zero ID number. Because we are reading the ID and the amount owed with the same `scanf()`, a second number must be entered after the sentinel value to "satisfy" the format. Our instructions say to enter two zeros, but the code does not test the second number.
- On each repetition of the loop, we read one data value directly into `&x[k]`, the `k`th slot of the array `x`. At the end of each repetition, we increment `k` to prepare for processing the next array element. Each time around the loop the variable `k` contains a different value between 0 and `n-1`; after `n` iterations, data fill the first `n` array slots and the remaining slots still contain garbage.
- After the sentinel value has been read, we exit from the loop. At this time, the value of `k` is the number of real data items that have been read and stored, excluding the sentinel. We return this essential information to the caller, which will store it and use it to control all future processing on these arrays.
- When control returns to the caller, it can use the values stored in the array by the function.

Notes on Figures 10.20 and 10.21. Sequential search. This function assumes that the data are unsorted and that all items must be checked before we can conclude that the search has failed. Two versions are given, a simpler one with two return statements and a longer one with a status flag.

The function header. The parameters include elements required for a search algorithm: a table, the number of items to be searched, and a search key. The table is a simple integer array containing `n` values. The return value will be a failure code or the subscript of the key value in the array if it exists.

In the bill-payment application we use a pair of parallel arrays containing a list of account numbers and the amount owed on each account. Most actions taken by the program (input, output) involve both arrays. However, the arrays are treated differently when we search. Only the index column, in this case the ID array, is passed to the search function.

A simple sequential search function for an unsorted table.

```
int sequential_search( int ID[], int n, int key )
{
    int cursor;      /* Loop counter and array index */

    for (cursor = 0; cursor < n; ++cursor) {
        if ( ID[cursor] == key ) return cursor;
    }

    return -1;
}
```

Figure 10.20. Sequential search of a table.

An alternative way to code a search function that uses only one return statement. To accomplish this goal, we use a status flag.

```
int sequential_search( int ID[], int n, int key )
{
    int cursor;      /* Loop counter and array index */
    int found = 0;   /* False now, becomes true if key is found.  */

    for (cursor = 0; !found && cursor < n; ++cursor) {
        if ( ID[cursor] == key ) found = 1;   /* true */;
    }

    return (found ? cursor-1 : -1);
}
```

Figure 10.21. Searching without a second return statement.

Style. Some experts believe that programmers should never use two return statements in a function. This slightly longer version of the search loop does the same job by using a status flag in place of the second return statement.

First box: the status flag. To avoid using either a `break` statement or a second `return` statement, we introduce a status flag named `found`. This is initialized to false (0) and will be set to true (1) within the search loop if the key item is found.

Second box: the search loop. A counted loop is used to examine the data items. If a match is found for the key, the loop terminates; otherwise, all `n` values are checked.

First inner box: the comparison. Since the base type of the array is `int`, we use the `==` operator to compare each item to the key value. The search succeeds if the result is true (1).

Second inner box: Dealing with Success. If a match is found, we need to leave the loop. This is done in different ways by our two versions of this function.

- Sometimes we abort a loop with a `break` statement. Here, we use `return` for the same purpose. It causes control to leave both the loop and the function. The current item's subscript is returned to the caller.
- We avoid using a second return statement tests by setting a status flag to true (1), indicating that the key value has been found. Control does not leave the loop. It returns to the top of the loop and increments `cursor`, making it one too large. The function return statement will have to compensate for this extra increment.

Last box, Figure 10.20: Failure. If the loop goes past the last data item without finding a match, the search has failed. Failure is indicated by returning a subscript of -1, a subscript that is invalid for any array, and if often used to indicate error or failure.

Last box, Figure 10.21: Returning. This single return statement must handle both success and failure. If the item was not found, we want to return -1 to indicate failure. If it was found, we must return the value of `cursor`-1. The value of `cursor` is too largeby 1 because the `for` loop incremented it after `found` was set to true and before `found` was tested to terminate the loop.

The little-used conditional operator provides a way to use a single return statement to return one thing or another. It works like an `if...else` except that it is an expression, not a statement. Read the statement like this: "If `found` is true, then return `cursor-1` else return -1".

10.7 The Maximum Value in an Array

Finding the maximum value in an array is an easy but nontrivial task. The `find_max()` function presented here scans the data array sequentially, like a search function, but it does not use a search key. To illustrate this algorithm, we use the same parallel-array data structure that was used in Figure 10.18, and search the data for the largest unpaid bill.

Notes on Figure 10.22. Who owes the most?

First box: Prototypes. The `get_data()` function is in Figure 10.20 and `find_max` is in Figure 10.23.

Second box: data declarations. We are using the same data structure as in the sequential search application: a set of parallel arrays containing the ID numbers and unpaid balances of a set of up to `ACCOUNTS` customers. We need `n` to store the actual number of customers that were input and `where` to store the position of the customer with the largest bill.

```
#include <stdio.h>
#define ACCOUNTS 20

int  get_data( int ID[], float owes[], int nmax );
int  find_max( int ID[], int n );

int main( void )
{
    int n;                    /* #of ID items; will be <=ACCOUNTS.    */
    int ID[ ACCOUNTS ];       /* ID's of members with overdue bills.  */
    float owes[ ACCOUNTS ]; /* Amount due for each member.          */
    int where;                /* Position of maximum value.           */

    n = get_data( ID, owes, ACCOUNTS );   /* Input all unpaid bills. */
    printf( "\nLargest Unpaid Account:\n" );

    where = find_max( owes, n );
    printf( "\tID# %i owes $ %.2f\n", ID[where], owes[where] );

    return 0;
}
```

Figure 10.22. Who owes the most? Main program for finding the maximum.

```
int                                /* Find the maximum value in an array.  */
find_max( int data[], int n )
{
    int finger = 0;                /* Put your finger on the first value. */
    int cursor;

    for (cursor = 1; cursor < n; ++cursor) {
        if (data[finger] < data[cursor])   /* If you find a bigger value...*/
            finger = cursor;               /* ...move your finger to it.   */
    }

    return finger;                 /* Your finger is on the biggest value. */
}
```

Figure 10.23. Finding the maximum value.

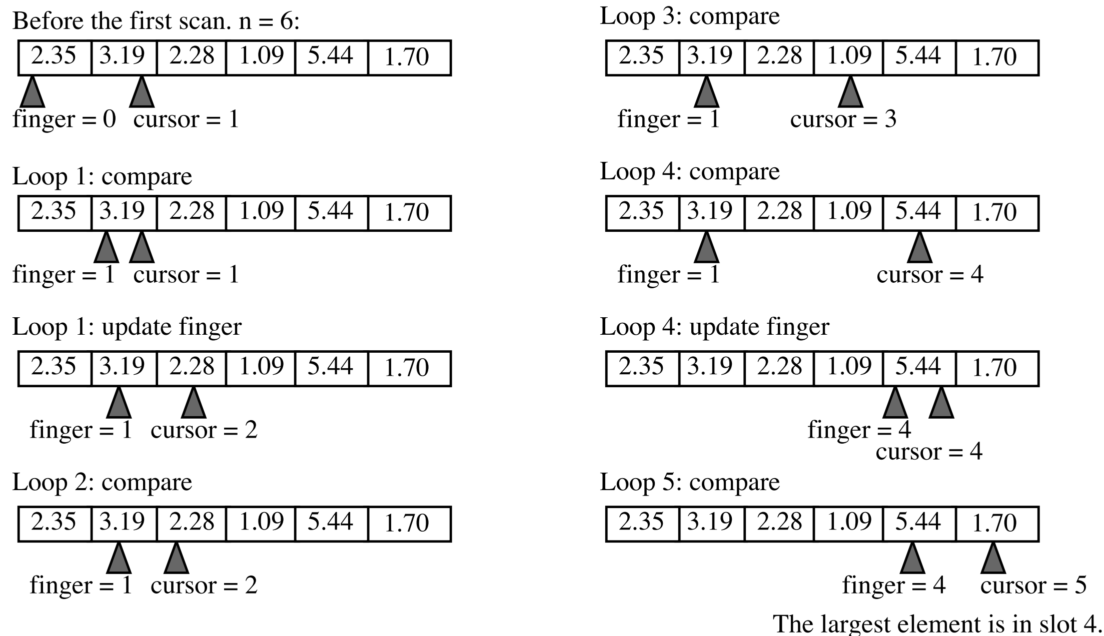


Figure 10.24. The maximum algorithm, step by step.

Third box: doing the work. We call the `find_max()` function in Figure 10.23 to search for the largest value in the array `owes` which contains `n` data items. The result is stored in `where`, and is then used to print out both the biggest bill and the ID number of the customer who owes the most.

Notes on Figure 10.23. Finding the maximum value.

The function header. The `find_max()` function is called from `main()` in Figure 10.22. We only need two parameters: the array to search and the length of that array.

First box: initialization. The idea is to scan the array sequentially, but at all times, to keep one "finger" on the biggest value we have seen so far. To get started, we set `finger` to slot 0.

Second box: the search loop. To find the largest value in an array, we must examine every array element. We start by designating the first value as the biggest-so-far, then scan start a sequential scan from array slot 1, looking for something bigger. Whenever we find the value under the `cursor` is bigger than the one under the `finger`, we update `finger`. When the loop ends, `finger` will be the position of the largest value. This process is illustrated in Figure 10.24.

10.8 Sorting by Selection

A common operation performed on arrays is sorting the data they contain. Many sorting methods have been invented: Some are simple, some complex, some efficient, some miserably inefficient. In general, the more complex sorting algorithms are the most efficient, especially if the array is very long.

In this section, we look at **selection sort**, which can be used on a small number of items. It is one of the simplest sorting methods, but also one of the slowest. Nonetheless, selection sort has the advantage that, if you stop in the middle of the process, one part of the array is fully sorted, so it is a reasonable way to find the either the largest or smallest few items in a long array.[7]

The basic selection strategy has several variations: the data can be sorted in ascending or descending order, the work can be done by using either a maximum or a minimum function, and the sorted elements can be collected at either the beginning or the end of the array. In this section, we develop a version that sorts the array elements in ascending order, by using a maximum function and collecting the sorted values at the end of the array. At any time, the array consists of an unsorted portion on the left (initially the whole array) and a sorted portion on the right (initially empty). To sort the array, we make repeated trips through smaller and smaller portions of it. On each trip, we locate the largest remaining value in the unsorted part of the array, then move it to the beginning of the sorted area by swapping it with whatever value happens to be there. After k trips, the k largest items have been selected and placed, in order, at the end of the array. After $n - 1$ trips, the array is sorted. This process is illustrated in Figure 10.25.

A program to implement this algorithm is developed easily by a top-down analysis. A problem specification is given in Figure 10.26.

10.8.1 The Main Program

A well-designed main program is like an outline of the process; it calls on a series of functions to do each phase of the actual job. This kind of design is easy to plan, easy to read, and easy to debug. The sorting task has three major phases: input (read the numbers), processing (sort them), and output (print the sorted list). For each phase, the main program should call a function to do the job and display a comment that reports the progress. Programs that contain arrays and loops often take a while to debug; during that time, generous feedback helps the programmer identify the location and nature of the errors.

We start by writing the obvious parts of `main()`, borrowing elements from the previous programs (sequential search and finding the maximum), where possible. Much can be borrowed, including

- Prototypes for the I/O functions that work with the parallel arrays (`get_data()` and `print_data()`,
- The skeleton of `main()`,
- Declarations within `main()` for the parallel-array data structure.
- Since the selection sort algorithm must find the maximum value in an array, we also include the prototype for `find_max()`.

[7] A simple algorithm, **insertion sort**, has been shown to be the fastest sort of all when used on short arrays (fewer than 10 items). We present this algorithm in Chapter 17. The **quicksort** (discussed in Chapter 19) is a much better way to sort moderate length and long arrays. Two other simple sorts, bubble sort and exchange sort, have truly bad performance for all applications. They are not presented here and should not be used.

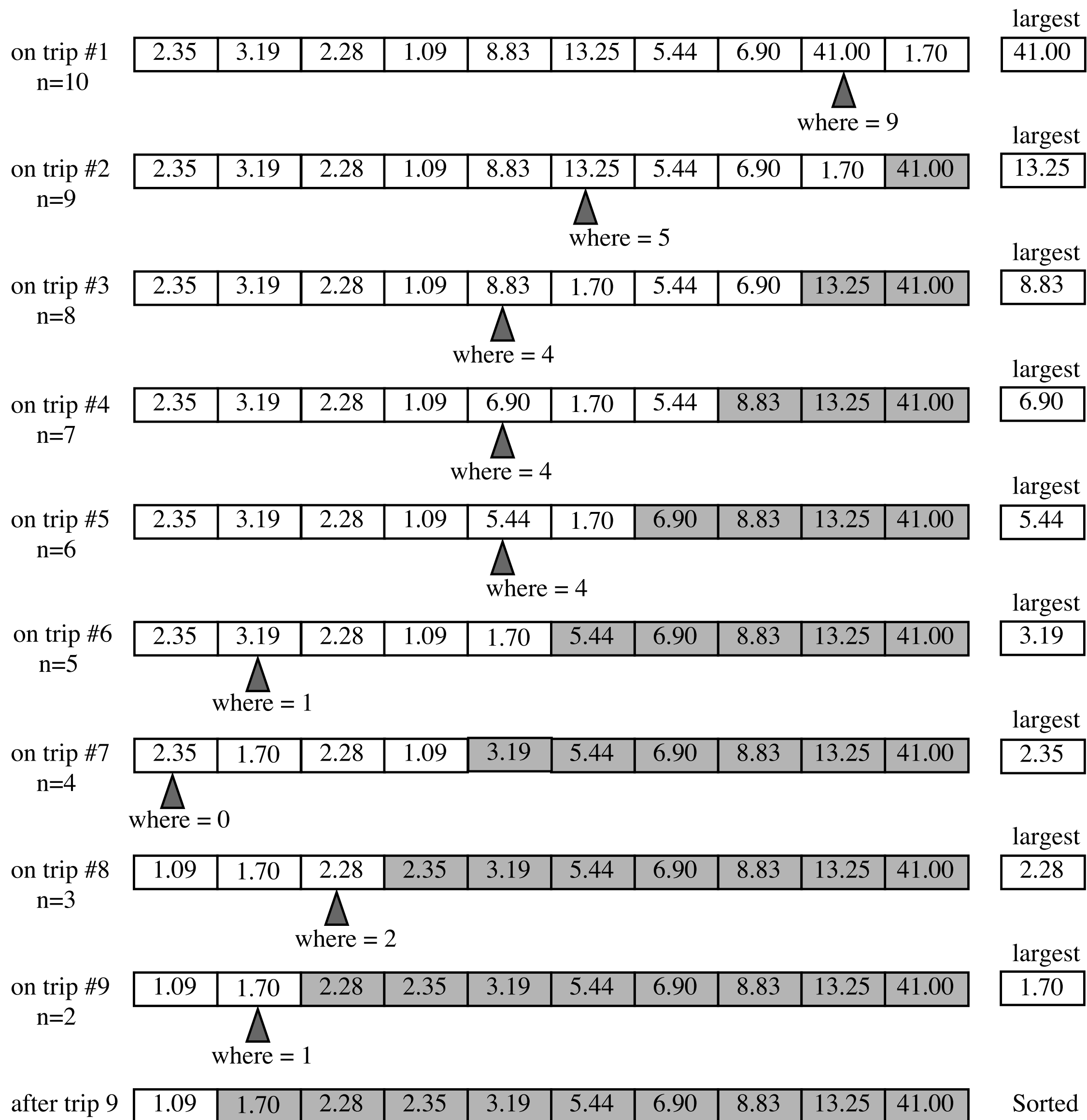


Figure 10.25. The selection sort algorithm, step by step.

Goal: Sort customer billing records in ascending order according to the amount owed. These records are stored in a parallel-array data structure with two columns: ID number and amount owed. There are `n` records altogether.

Input: ID numbers and amounts due for a set of customers.

Output: A list of customer records, in order, by the amount owed.

Algorithm: Use a selection sort, as follows:
Repeat the following actions $n - 1$ times:

1. Let `where` be the subscript of the maximum-valued element in the array between subscripts 0 and `n-1`.
2. Swap the element at position `where` with the element at position `n-1`. The swapped value will now be in its proper sorted position.
3. Decrement `n` to indicate that there are now fewer unsorted items.

Figure 10.26. Problem specifications: Sorting the Billing Records

Step 1: The obvious necessities.

```
#include <stdio.h>
#define ACCOUNTS 20

int  get_data( int ID[], float owes[], int nmax );
print_data( ID, owes, n );      /* Print final amounts owed. */
find_max( int ID[], int n );    /* Located largest key value in array. */

int main( void )
{
    int n;                   /* #of ID items; will be <=ACCOUNTS.    */
    int ID[ ACCOUNTS ];      /* ID's of members with overdue bills.  */
    float owes[ ACCOUNTS ];  /* Amount due for each member.          */
    ...
    return 0;
}
```

Step 2: Input. At the position of the dots, we add calls on the input and output functions, following the example of Figure 10.18.

```
n = get_data( ID, owes, ACCOUNTS );   /* Input all unpaid bills. */
printf( "\nInitial List of Unpaid Bills:\n     ID  Amount\n" );
print_data( ID, owes, n );            /* Echo the input */
```

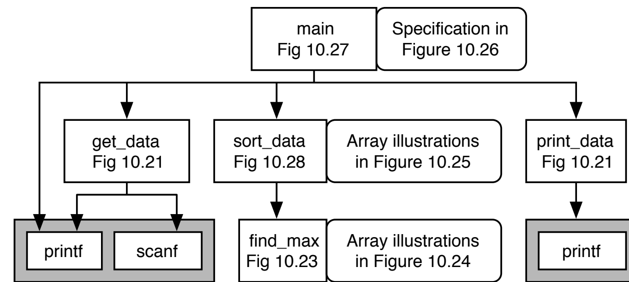


Figure 10.27. Call chart for selection sort.

Step 3: Processing. We invent the name `sort_data()` for the selection sort function. In general, the function needs to know what array to sort and the length of that array. It rearranges the data within the array and returns the sorted values in the same array, with no need for any additional return value. In this case, we are sorting a pair of parallel arrays, so both arrays must be rearranged in parallel. Thus, the `sort_data()` function must take both arrays as parameters. We write a prototype (at the top) and a call for this function inside `main()`:

```
void sort_data( int data[], float key[], int n );
...
sort_data( ID, owes, n );
```

Output. When sorting is finished, we need to output the sorted data. So we add another call on `print_data()` at the end of `main()`:

```
print_data( ID, owes, n );            /* The sorted data */
```

A call chart for the overall program is given in Figure 10.27. The completed `main()` function is shown in Figure 10.28.

10.8.2 Developing the `sort_data()` Function

We must start with a thorough understanding of the algorithm. This is illustrated in Figures 10.24 and 10.25 and defined carefully in Figure 10.26. Once the method is understood, we are ready to implement the `sort_data()` function. (The code fragments that follow are assembled in Figure 10.29.) We begin with the function skeleton and declare the variables mentioned in the problem specifications.

```
void sort_data( int data[], float key[], int n );
{
    int where; ...
}
```

This program calls the sort function in Figure 10.29, and the input and output functions from Figure 10.20.

```
#include <stdio.h>
#define ACCOUNTS 20

int get_data( int ID[], float owes[], int nmax );
void print_data( int ID[], float owes[], int n );
void sort_data( int data[], float key[], int n );

int main( void )
{
    int n;                  /* #of ID items; will be <=ACCOUNTS.    */
    int ID[ ACCOUNTS ];     /* ID's of members with overdue bills.  */
    float owes[ ACCOUNTS ]; /* Amount due for each member.          */

    n = get_data( ID, owes, ACCOUNTS );   /* Input all unpaid bills. */
    printf( "%i items were read; beginning to sort.\n", n );
    sort_data( ID, owes, n );

    puts( "\nData sorted, ready for output" );
    print_data( ID, owes, n );
    return 0;
}
```

Figure 10.28. Main program for selection sort.

The loop skeleton and body. Step 2 of the specification calls for a process to be repeated $n-1$ times to sort n things. We write a `for` loop that implements this control pattern:

```
for (start=n-1; start>0; --start) {
    ...
}
```

Now we need to write the body of the loop. We have a function that finds the maximum value in an array, so we call it and save the result.

```
where = find_max( key, n );
```

Next, we must swap the large value at position `where` with the last unsorted value in the array, which is at position `start`. Swapping takes three assignments, but since we are working on a parallel-array data structure, identical swaps must be made on both arrays, for a total of six assignments. In the code below, the instructions on the left swap the key array, those on the right swap the data array.

Also, on each repetition, we start at the high-subscript end of the unsorted array, that is, at slot $n-1$.

This function is called from main() in Figure 10.28; it calls find_max() in Figure 10.23.

```
int find_max( float data[], int n );   /* Prototype not included by main(). */

void sort_data( int data[], float key[], int n )
{
    int start;         /* End of unsorted data in the array. */
    int where;         /* Position of largest value in the key array. */
    int bigData;       /* For swapping the data column. */
    float bigValue;    /* For swapping the key column. */

    for (start=n-1; start>0; --start) {
        where = find_max( key, n );
        /* Swap the two columns of the table in parallel. */
        bigValue = key[where];      bigData = data[where];
        key[where] = key[start];    data[where] = data[start];
        key[start] = bigValue;      data[start] = bigData;
        --n;
    }
}
```

Figure 10.29. Sorting by selecting the maximum.

On each repetition, one item is placed in its final position, leaving one fewer item to be sorted. So we decrement n just before the end of the loop.

```
bigKey = key[where];        bigData = data[where];
key[where] = key[start];    data[where] = data[start];
key[start] = bigKey;        data[start] = bigData;
--n;
```

This completes the algorithm and the program. We gather all the parts of sort_data() together in Figure 10.29.

Testing. We combined all of the pieces of the sort program, compiled it, and ran it on the file sele.in, which contains the data listed in Figure10.30. The input is on the left and the corresponding output on the right.

Input Phase		Output Phase
Enter pairs of ID and unpaid bill (two zeros to end):		Data sorted, ready for output
31	2.35	[0] 13 1.09
7	3.19	[1] 25 1.70
6	2.28	[2] 6 2.28
13	1.09	[3] 31 2.35
22	8.83	[4] 7 3.19
38	13.25	[5] 19 5.44
19	5.44	[6] 32 6.90
32	6.90	[7] 22 8.83
3	41.	[8] 38 13.25
25	1.70	[9] 3 41.00
0	0	

Figure 10.30. Input and output for selection sort.

10.9 What You Should Remember

10.9.1 Major Concepts

Arrays and their use. An array in C is a collection of variables stored in order, in consecutive locations in the computer's memory. The array element with the smallest subscript (0) is stored in the location with the lowest address. We use arrays to store large collections of data of the same type. This is essential in three situations:

- When the individual data items must be used in a random order, as with the items in a list or data in a table.
- When each data value represents one part of a compound data object, such as a vector, that will be used repeatedly in calculations.
- When the data must be processed in separate phases, as in the problem from Figure 10.17. In this program, all the account balances must first be read and stored. Then the stored data must be searched and updated, using bill-payment amounts.

Parallel arrays. A multicolumn table can be represented as a set of parallel arrays, one array per column, all having the same length and accessed using the same subscript variable. Multidimensional arrays also exist and are discussed in Chapter 18.

Array arguments and parameters. An array name followed by a subscript in square brackets denotes one array element whose type is the base type of the array. This element can be used as an argument to a function that has a parameter of the base type. To pass an entire array as an argument, write just the

array name with no subscript brackets. The corresponding formal parameter is declared with empty square brackets. When the function call is executed, the address of the beginning of the array will be passed to the function. This gives the function full access to the array; it can use the data in it or store new data there.

Array initializers. C allows great flexibility in writing array initializers; we summarize the rules here:

- An array initializer is a series of constant expressions enclosed in curly brackets. These expressions can involve operators, but they must not depend on input or run-time values of variables. The compiler must be able to evaluate the expressions at compile time.
- If there are too many initial values for the declared length, C will give a compile-time error comment.
- If there are too few initial values, the uninitialized areas will be filled with 0 values of the proper type: an integer 0, floating value 0.0, or pointer `NULL`.
- The length of an array may be omitted from the declaration if an initializer is given. In this case, the items in the initializer will be counted and the count will be used as the length of the array.

Searching. Many methods can be used to search an array for a particular item or one with certain characteristics. A sequential search starts at the beginning of a table and compares a key value, in turn, to every element in the key column. The search ends when the key item is found or after each item has been examined.

The typical control structure for implementing a sequential search is a `for` loop that moves a subscript from the beginning of the array to the end. In the body of this loop is an `if...break` statement that compares the key value to the current table element.

The search can either succeed (find the key value) and break out of the loop or fail (because the key value does not match any item in the table). A sequential search for a specific item is slow and appropriate for only short tables. It is slightly more efficient when the table is in sorted order, because failure can be detected prior to reaching the end of the table. However, binary search (see Chapter 19) is an even faster algorithm for use with sorted data.

If the data are sorted according to the search criterion, shortcuts may be possible. However, a sequential search is necessary when the order of the data in the array is unrelated to the criterion because all the data items must be examined. For example, finding the longest word in a dictionary would require looking at every word (a sequential search) because a dictionary is sorted alphabetically, not by word length.

Sorting. Locating a particular item in a table can be done much more efficiently if the information is sorted. Many sort algorithms have been devised and studied; among the simplest (and slowest) is the selection sort. It sorts n items by selecting the minimum remaining element $n-1$ times and moving it to a part of the array that will not be searched again. Other more efficient techniques such as the insertion sort (Chapter 17) and the quicksort (Chapter 19) are examined later.

10.9.2 Programming Style

Usage.

- Use a defined constant to declare the length of an array. This way the use and the array declarations will be consistent and easily changed if the need arises.

- A `for` loop typically is used to process the elements of an array. The values of the loop counter go from 0 to the length of the array (which is given by a defined constant or a function parameter); for example, `for (k=0; k<N; ++k) printf( "%g ", volume[k] );`. Note that this loop paradigm stops before attempting to process the nonexistent element `volume[N]`.

Names. Variable names such as `j` and `k` typically are used as array subscripts since they are commonly found in mathematical formulas. However, when writing a program that sorts, it is very helpful to use meaningful names for the subscript variables. You are much more likely to write the code correctly in the first place, and then get it debugged, if you use names like `cursor` and `finger` rather than single-letter variable names such as `i` and `j`.

Local vs. global. Constants should be declared globally if they are used by more than one function or if they are purely arbitrary and likely to be changed. If a constant is used by only one function, it may be better to declare it locally. However, a large set or table of such constants will incur large setup times each time the function is called. These constants should be declared as `static const` values, which are only initialized once.

Don't talk to strangers. Each object name used in a function should represent an object that fits into one of the following categories:

- A global constant, `#define`d at the top of the program, or like `NULL`, in a header file.
- A parameter, declared and named in the function header.
- A local variable or constant, declared and named within the function (an object should be local if it is used only within a function and does not carry information from one function to another).
- A global function whose prototype is at the top of the program or in a header file.
- A local function, declared within the function and defined after it (a function should be declared locally if it is used only within that function. This does not cause any run-time inefficiency).

Modularity. We wrote a `sort_data()` function as a loop that calls the `find_min()` function. Within that function is another loop. When written like this, the logic of the program is completely transparent and easily understood. In many texts, this algorithm is written as a loop within a loop. This second form takes fewer lines of code and executes more efficiently, because no time is spent calling functions. However, it is not so easy to understand. Which form is better? The modular form. Why? Because it can be debugged more easily and is less likely to have persistent bugs. Doesn't efficiency matter? It often does, but if so, a better algorithm (such as quicksort) should be used instead. It is a false economy to use bad programming style to optimize a slow algorithm.

Sorted vs. unsorted. If the data we wish to search already are sorted, by all means we should take advantage of this. If not, we need to decide whether to sort the data before searching. This issue will be addressed to some extent in later chapters. However, it is a complex issue involving the data set size, how fast the data set changes, which data structures and algorithms are used, and how many times a search will

be performed. The general topic of data organization and retrieval is the subject of dozens of books on data structures and databases.

Software reuse. Do not waste time trying to reinvent the wheel. If a library routine meets your need, use it. If you have previously written a function that does almost what you need, modify it as necessary. If someone else has developed a solution for a certain task, such as sorting, go ahead and use it, after you have verified that any assumptions it makes are satisfied by your data and structures.

10.9.3 Sticky Points and Common Errors

Array length vs. highest subscript. The number given in an array declaration is the actual number of slots in the array. Since array subscripts start at 0, the highest valid subscript is one less than the declared length. Often, though, there are more slots than valid data. This happens during an input operation and whenever the total amount of data entered falls short of the maximum allowed. In such situations, another variable is used to store the number of actual data elements in the array.

Subscript errors. Programmers accustomed to other languages often are surprised to learn that C does absolutely no subscript range checking. If a subscript outside the defined range is used, there will be no error comment from the compiler or at run-time. The program will run and simply access a memory location that belongs to some other variable. For example, if we write a loop to print the values in an array and it loops too many times, the program starts printing the values adjacent to the array in memory. At best, this results in minor errors in the results; at worst, the program can crash.

Caution: do not fall off the end of an array. Remember that C does not help you confine your processing to the array slots that you defined. When you use arrays, avoid any possibility of using an invalid subscript. Input values must be checked before using them as subscripts. Loops that process arrays must terminate when the loop counter reaches the number of items in the array.

Ampersand errors. Arrays and nonarrays are treated differently in C. An array argument always is passed to a function by address. We do not need to use an ampersand with an unsubscripted array name.

Array parameters. To pass an entire array as an argument, write just the name of the array, with no ampersands or subscript brackets. The ampersand operator is not necessary for an array argument because the array name automatically is translated into an address. The corresponding array parameter is declared with the same base type and empty square brackets. A number can be placed between the brackets, but it will be ignored.

Array elements as parameters. A single array element also can be passed as a parameter. To do this, write the array name with square brackets and a subscript. If the function is expected to store information in the array slot, as `scanf()` might, you must also use an ampersand in front of the name.

Sorting. Writing a sorting algorithm can be a little tricky. It is quite common to write loops that execute one too many or one too few times. When debugging a sort, be sure to examine the output closely. Check that the items at the beginning and end of the original data file are in the sorted file and that the output has the correct number of items. Examine the items carefully and make sure all are there and in order. It is common to make an error involving the first or last item. Test all programs on small data sets that can be thoroughly checked by hand.

Parallel arrays. A table can be implemented as a set of parallel arrays. When sorting such a table, it is important to keep the arrays all synchronized. If the items in one column are swapped, be sure to swap the corresponding items in all other columns. Using an array of structures may solve this problem, but this solution may have its own drawbacks, which we have discussed previously.

10.9.4 Where to Find More Information

- Arrays of strings are presented in Chapter 12, arrays of structures in Chapter 13 and arrays of functions in Chapter 17.
- Dynamic allocation of arrays and arrays of pointers are found in Chapter 16.
- Pointers are introduced in Chapter 11 and the use of pointers to process arrays is explained in Chapter 17.
- Two dimensional arrays and their applications are covered in Chapter 18. Multidimensional arrays and arrays of pointers to arrays are in the same chapter.
- Other sorting algorithms are presented in later chapters. Insertion sort is in Chapter 17; quicksort in Chapter 19.
- Other array algorithms presented are: Binary search: Chapter 19, Shuffling a deck: Chapter 14, Gaussian elimination: Chapter 18, Simulation: Chapter 16.

10.9.5 New and Revisited Vocabulary

These are the most important terms and concepts presented in this chapter:

aggregate type	constant expression	status flag
array	parallel arrays	sequential search
base type	effective address	search loop
array slot	walking on memory	key column
array element	memory error	data column
array length	array argument	search key
size of an array	array parameter	sorted table
subscript	sequential array processing	position variable
array declaration	prime number	finding the minimum
array initializer	divisibility	selection sort

10.10 Exercises

10.10.1 Self-Test Exercises

1. Which occupies more memory space, an array of 15 `short int`s or an array of 3 `double`s? Explain your answer.
2. An array will be used to store temperature readings at four-hour intervals for one day. It is declared thus: `float temps[6];`
   (a) Draw an object diagram of this array.
   (b) What is its base type? Its length? Its size?
   (c) Write a loop that will read data from the keyboard into this array.
   (d) Write an `if` statement that will print `freezing` if the temperature in the last slot is less than or equal to 32°F and `above freezing` otherwise.
3. Array and function declarations.
   (a) Write a declaration with an initializer for the array of `float`s pictured here.

ff

1.9	2.5	-3.1	17.2	0
[0]	[1]	[2]	[3]	[4]

   (b) Write a complete function that takes this array as a parameter, looks at each array element, and returns the number of elements greater than 0.
   (c) Write a prototype for this function.
   (d) Write a `scanf()` statement to enter a value into the last slot of the array.
4. In the indicated spots below, write a prototype, function header, and call for a function named `CHKBAL` that computes and returns the balance in a checking account. Its parameters are an initial account balance and an array of check amounts. You need not actually write a whole function to compute the new account balance; just fill in the indicated information.

```
#define  X  5
/* insert prototype here */

int main( void )
{
    float check_amounts[X];   /* $ amounts of checks */
    float start_balance;      /* balance before checks */
    float end_balance;        /* balance after checks */
    /* put call here */
}
/* put function header here */
```

5. The following are two declarations and a `while` loop.

```
int ara[13] = {1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096};
int k = 12;
while (k > 0) {
    printf( "%2i: %i \n", k, ara[k] );
    k -= 2;
}
```

   (a) What is the output?
   (b) Rewrite the code using a `for` loop instead of the `while` loop.

6. Given the following declarations, the prototypes on the left, and the calls on the right, answer *good* if a function call is legal and meaningful. If there is an error in the call, show how you might change it to match the prototype.

```
int j, a[5];
float f, flo ;
double x, dub[4];
```

	Prototype	Call
(a)	`double fun( double d[] );`	`x = fun( dub[] );`
(b)	`void fill( double d );`	`x = fill( dub );`
(c)	`int fix( float f ) ;`	`fix( flo[5] );`
(d)	`int hack( int a[] );`	`j = hack( a );`
(e)	`double q( double d[], int n );`	`x = q( dub[0], a[0] );`

7. Use the following definitions in this problem:

```
#define  LIMIT 5
int j;
double load[LIMIT];
```

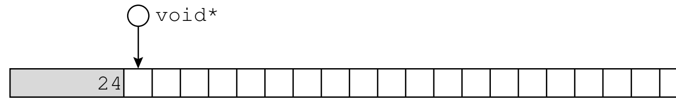
   (a) Draw an object diagram for the array named `load`; identify the slot numbers in your drawing. Also show the values stored in it by executing the following loop:

```
for ( j=0; j<LIMIT; j++ ) {
    if ( j % 2 == 0 ) load[j] = 10.2 * (j + 1);
    else load[j] = (j + 1) * 2.5;
}
```

   (b) Trace the following loop and show the output exactly as it would appear on your screen. Show your work.

```
for (j = 0; j < LIMIT; ++j) {
    if (j <= LIMIT / 2)  printf( "\t%6.3g", load[j] );
    else printf( "\t%5.2f", load[j] -.05 );
}
```

8. The following diagram shows an array of odd integers. Declare and initialize a parallel array of type `int` that contains 1 (true) in the slot corresponding to every prime number, and 0 (false) for the nonprime numbers.



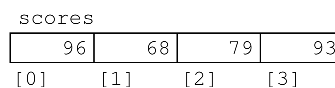
9. Consider an array containing six data items that you must sort using the selection sort algorithm. How many data comparisons does the algorithm perform in the `find_min()` function? How many times is `find_min()` called, and how many comparisons are made (total) during all these calls? How many (total) data swaps does `sort_data()` perform?
10. Suppose you are searching for an item in a sorted array of N items. Using a sequential search algorithm, how many items are you likely to check before you find the right one? Is this number the same whether or not the item is present in the array? Express the answer as a function of N.
11. The selection sort in the text generated a list of values in ascending order. How would you change the algorithm to generate numbers in descending order?
12. Consider the following list of numbers: 77, 32, 86, 12, 14, 64, 99, 3, 43, 21. Following the example in Figure 10.23, show how the numbers in this list would be sorted after each pass of the selection sort algorithm.

10.10.2 Using Pencil and Paper

1. Draw a flow diagram for the main program in Figure 10.15 and for the `divisible()` function in Figure 10.16 (you may omit the details of the other function). In your diagram, show how control flows from one function to the other and back again.
2. An array will be used to store the serial numbers of the printers in the lab. It is declared thus:
   `long printer_ID[10];`

   (a) Draw an object diagram of this array.
   (b) What is its base type? Its length? Its size?
   (c) Write a loop that will read data from the keyboard into this array; the loop should end when the user enters a negative number. Store the actual number of data items in the variable named `count`.
   (d) Write a loop that will print out all `count` data items from the array.

3. Array and function declarations.
   (a) Write a declaration with an initializer for the array of small integers pictured here:



(b) Write a complete function that takes this array and an integer `n` as parameters, tests each array element, and prints all array elements greater than `n`.

(c) Write a prototype for this function.

4. Use the following definitions in this problem:

```
#define  LIMIT 7
int j;
short puzzle[LIMIT], crazy[LIMIT];
```

(a) Draw an object diagram for the array named `puzzle`; identify the slot numbers in your drawing. Also show the values stored in it by executing the following loop. This loop performs an illegal operation. Your job is to figure out what will happen and why.

```
for (j = LIMIT; j > 0; --j) {
    if (j+4 > j*2) {
        puzzle[j] = j*2;
        crazy[LIMIT-j] = j + 2;
    }
    else {
        puzzle[j] = j/2;
        crazy[LIMIT-j] = j - 2;
    }
}
```

(b) Trace the following loop and show the output exactly as it would appear on your screen. Show your work.

```
for (j = 0; j < LIMIT; ++j) {
    if (j % 2 == 1)
        printf( "\t%5i %3i", puzzle[j], crazy[j] );
    else  printf( "\t%3i %5i", crazy[j], puzzle[j] );
}
```

5. The following are two declarations and a `for` loop.

```
int a[12] = {2, 4, 8, 3, 9, 27, 4, 16, 64, 5, 25, 125};
int k;
for (k = 1; k < 10; k += 2) printf("%i: %i \n", k, a[k]);
```

(a) What is the output?

(b) Rewrite the code using a `while` loop instead of the `for` loop.

6. Given the following declarations, the prototypes on the left, and the calls on the right, answer *good* if a function call is legal and meaningful. If there is an error in the call, show how you might change it to match the prototype.

```
int j, ary[5];
float f, flo ;
double x, dub[4];
```

(a) `double list( double d[] );` `x = list( ary );`
(b) `void handle( int k );` `handle( ary[5] );`
(c) `void bank( float f, int n );` `bank( flo[5], j );`
(d) `int days( int a[] );` `j = days( &flo );`
(e) `int area( double d );` `x = area( dub[0] );`

7. In the spots indicated below, write a prototype, function header, and call for a function named `MISSING`. Its parameters are an array of student assignment scores and the number of assignments that have been graded and recorded. The function will look at the data and return the number of nonzero scores in the array. You need not actually write the whole function, just fill in the indicated information.

```
#define  MAX  10
/* insert prototype here */

int main( void )
{
    int assignments[MAX];   /* grades */
    int actual;             /* # of assignments so far. */
    int done;               /* # of nonzero grades. */
    /* put call here */
}

/* put function header here */
```

8. An unsuccessful search for an item in sorted and unsorted data arrays will require different numbers of comparisons. Compare a sequential search on these two types of data and explain why they are different (in terms of the number of comparisons performed).
9. Modify the code in the sequential search function in Figure 10.20 so that it assumes the data in the array are sorted in descending order. Do not search any more positions than necessary. Still return a value of `-1` if the key value cannot be found.
10. Consider an array containing N data items that you must sort using the selection sort algorithm. How many data comparisons does the algorithm perform? How many data swaps does it perform? Express the answer as a function of N.
11. Consider the following list of numbers: 77, 32, 86, 12, 14, 64, 99, 3, 43, 21. Following the example in Figure 10.25, show how the numbers in this list would be sorted after each pass of a selection sort algorithm that sorts numbers in *descending* order.

10.10.3 Using the Computer

1. Seeing bits.
   The program in Figure 4.23 shows how to convert an integer to any selected base. Obviously, this program works for base 2, binary notation. Modify this program so that it inputs a number and prints the equivalent value in binary notation. Do not print it in the expanded form of the previous program, but as a series of ones and zeros. You will need to store the binary digits in an array and print them later in the opposite order, so that the first digit generated is the last printed. For example, if the input were 22, the output should be 10110.

2. Global warming.
   As part of a global warming analysis, a research facility tracks outdoor temperatures at the North Pole once a day, at noon, for a year. At the end of each month, these temperatures are entered into the computer and processed. The operator will enter 28, 29, 30, or 31 data items, depending on the month. You may use −500 as a sentinel value after the last temperature, since that is lower than absolute 0. Your main program should call the `read_temps()`, `hot_days()`, and `print_temps()` functions described here:

   (a) Write a complete specification for this program.
   (b) Write a function, `read_temps()`, that has one parameter, an array called `temps`, in which to store the temperatures. Read the real data values for one month and store them into the slots of an array. Return the actual number of temperatures read as the result of the function.
   (c) Write a function, `hot_days()`, that has two parameters: the number of temperatures for the current month and an array in which the temperatures are stored. Search through the temperature array and count all the days on which the noon temperature exceeds 32°F. Return this count.
   (d) Write a function, `print_temps()`, with the same two parameters plus the count of hot days. Print a neat table of temperatures. At the same time, calculate the average temperature for the month and print it at the end of the table, followed by the number of hot days.

3. The tab.
   An office with six workers maintains a snack bar managed on the honor system. A worker who takes a snack records his or her ID number and the price on a list. Once a month, the snack bar manager enters the data into a computer program that calculates the monthly bill for each worker. No item at the snack bar costs more than $2, and monthly totals are usually less than $100.

   (a) Write a complete specification for this program.
   (b) Using a top-down development technique, write a main program that will call functions to generate a monthly report. These functions are described here. Declare an array of `float`s named `tabs` to store total purchase amounts for each member and the guests.
   (c) The `purchases()` function should have one parameter, the `tabs` array. This function should allow the manager to enter two data items for each purchase: the price and the ID number of the worker who made the purchase. The ID numbers must be integers between 1 and 6. In addition, the code 0 is used for guests, whose bills are paid by the company. As each purchase is read, the amount (in dollars and cents) should be added to the array slot for the appropriate worker. When the manager enters an ID code that is not between 0 and 6, it should be considered a sentinel value and a signal

to end the loop and return from the function. At that time, the array should contain the total purchases for each worker and for the guests.

(d) The `bills()` function should have one parameter, the `tabs` array. Print a bill for each worker, giving the ID number and the amount due.

4. Payroll.
The Acme Company has some unusual payroll practices and keeps the information in its personnel database in a strange way. The firm never has more than 200 employees and pays all its employees twice a month, according to the following rules:

(a) If the person is salaried, the pay rate will be greater than $1,000. There are 24 pay periods per year, so for one period, `earnings = payrate / 24`.

(b) If the person is paid hourly, the pay rate will be between $5 and $100 per hour and the earnings are calculated by this formula:
`earnings =  payrate * hours`.

(c) A pay rate less than $5 per hour or between $100 and $1,000 per hour is invalid and should be rejected.

(a) Write a complete specification for this program.

(b) Using a top-down development process and following the example of Figure 10.12, write a main program that prints the bimonthly payroll report. Since the number of employees changes frequently, `main()` should prompt for this information. Then call functions to perform the calculations and produce a report. Use the functions suggested here, and add more if that seems appropriate. You will need a set of parallel arrays to hold the ID number, pay rate, hours worked, and earnings for each employee.

(c) Following the example of Figure 10.21, write a function, `get_earnings()`. Read the data for all the employees from the keyboard into the parallel arrays.

(d) Write a function, named `earn()`, that uses the pay rate and hours worked arrays to calculate the earnings and fill in the earnings array.

(e) Write a function, named `pay()`, to calculate and return the earnings for one employee. If the pay rate is invalid, print an error comment and return 0.0.

(f) Write a function, named `payroll()`, that prints a neat table of earnings, showing all the data for each person. Also calculate the total earnings and print that value at the end of the list of employees.

5. Guess my weight.
At the county fair a man stands around trying to guess people's weight. You've decided to see how accurate he is, so you collect some data. These data are a set of number pairs, where the first number in the pair is the actual weight of a person and the second number is the weight guessed by the man at the fair. You decide to use two different error measures in your analysis: absolute error and relative error. Absolute error is defined as $E_{abs} = W_{guess} - W_{real}$, where W_{guess} and W_{real} are the guessed and real weights, respectively. The units of this error are pounds. The relative error is defined by $E_{rel} = 100 \times E_{abs}/W_{real}$, where the result of this equation is a percentage. Write a program that will input the set of weight pairs you accumulated, using a function with a sentinel loop to read the data.

The number of weight pairs should be between 1 and 100. Write another function that will calculate both the absolute and relative errors of the guesses and display them in a table. Finally, compute and print the average of the absolute values of the absolute errors and the average of the absolute values of the relative errors.

6. Having fen.
   Ms. Honeywell, an American businessperson, is preparing for a trip to Beijing, China, and is worried about keeping track of her money. She will take a portable computer with her, and wants a program that will sum the values of the Chinese fen (coins and bills) she has and convert the total to American dollars. Fen come in denominations of 1, 5, 10, 20, 50, 100, 200, 500, 1,000, and 2,000. The exchange rate changes daily and is published (in English) in the newspaper and on television. Write a program for Ms. Honeywell that will prompt her for the exchange rate and then for the number of fen she has of each denomination. Total the values of her fen and print the total as well as the equivalent in American dollars. Implement the table of fen values as a global constant array of integers.

7. A function defined by a table.
   Write a function that computes a tax rate based on earned salary according to the following table. In this function, if the salary value is not given exactly, use the rate for the next lower salary in the table. For example, use the rate 20% for $38,596.

Salary ($)	**Tax Rate (%)**
0	0
10,000	5
20,000	12
30,000	20
40,000	33
50,000	38
60,000	45
70,000	50

   Write a small program that will input a salary from the user, call your function to compute the tax rate, and then print the tax rate and the amount of tax to be paid. Validate the input salary so that it does not fall outside of the salary range in the table.

8. Moving average.
   Some quantities, such as the value of a stock, the size of a population, or the outdoor temperature, have frequent small fluctuations but tend to follow longer-term trends. It is helpful to evaluate such quantities in terms of a moving average; that is, the average of the most recent N measurements. (N normally is in the range 3...10.) This technique "smoothes out" the most recent fluctuations, exposing the overall trend. Write a program that will compute a moving average of order N for the price of a given stock on M consecutive days, where $M > N + 4$. To do this, first read the values of N and M from the user. Next read the first N prices and store them in an array. Then repeat the process below for the remaining $M - N$ prices:

   (a) Compute and print the average of the N prices in the array.

   (b) If all M values have been processed, quit and print a termination message.
   (c) Otherwise, eliminate the value in array slot 0 and shift the other $N - 1$ values one slot leftward.
   (d) Read a new value into the empty slot at the end of the array.

9. A bidirectional sort.
   Another sorting algorithm is similar to the selection sort, the "cocktail shaker" sort. This algorithm differs from the selection sort in the way it selects the next item from the array. Our selection sort always picks the maximum value from the remaining values and swaps it into the beginning of the sorted portion. For the cocktail shaker sort, the first pass finds the maximum data value and moves it to one end of the array. The second pass finds the minimum remaining value and moves it to the other end of the array. Subsequent passes alternate choosing the maximum and minimum values from the remaining data and moving that value to the appropriate end of the array. Eventually the two ends meet in the middle and the data are sorted. Write a program that implements the cocktail shaker sort just described and uses it to sort data sets containing up to 100 values.