Chapter 1
Arrays

Specific Learning Outcomes

The students must have the ability to:

1. Utilize the array data structure to represent lists and tables of values;
2. Define an array, initialize an array and refer to individual elements of an array;
3. Utilize arrays to store, sort and search lists and tables of values;
4. Define and manipulate multiple-subscripted arrays.

Learning Content

1. Defining Arrays;
2. Array Examples;
3. Passing Arrays to Functions;
4. Sorting Arrays;
5. Searching Arrays;
6. Multiple-Subscripted Arrays.

Arrays

An array is a group of contiguous memory locations that all have the same type. To refer to a particular location or element in the array, we specify the array's name and the position number of the particular element in the array. Figure 1 shows an integer array called c, containing 12 elements. Any one of these elements may be referred to by giving the array's name followed by the position number of the particular element in square brackets ([]). The first element in every array is the zeroth element (i.e., the one with position number 0). An array name, like other identifiers, can contain only letters, digits and underscores and cannot begin with a digit.

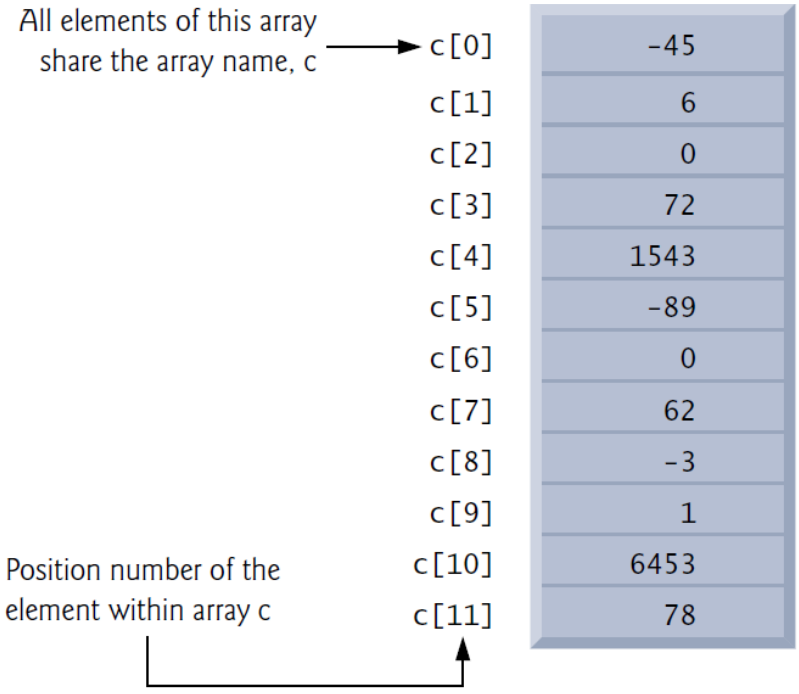

Figure 1. Array C

The position number in square brackets is called the element's index or subscript. An index must be an integer or an integer expression. For example, the statement c[2] = 1000; assigns 1000 to array element c[2]. Similarly, if a = 5 and b = 6, then the statement c[a + b] += 2; adds 2 to array element c[11]. An indexed array name is an lvalue—it can be used on the left side of an assignment.

Let's examine array c more closely. The array's name is c. Its 12 elements are referred to as c[0], c[1], c[2], …, c[10] and c[11]. The value stored in c[0] is –45, the value of c[1] is 6, c[2] is 0, c[7] is 62 and c[11] is 78. To print the sum of the values contained in the first three elements of array c, we'd write printf("%d", c[0] + c[1] + c[2]).

To divide the value of element 6 of array c by 2 and assign the result to the variable x, write x = c[6] / 2. The brackets used to enclose an array's index are actually considered to be an operator

in C. They have the same level of precedence as the function call operator (i.e., the parentheses that are placed after a function name to call that function). Figure 2 shows the precedence and associativity of the operators introduced to this point in the text.

Operators	Associativity	Type
[] () ++ *(postfix)* -- *(postfix)*	left to right	highest
+ - ! ++ *(prefix)* -- *(prefix)* (*type*)	right to left	unary
* / %	left to right	multiplicative
+ -	left to right	additive
< <= > >=	left to right	relational
== !=	left to right	equality
&&	left to right	logical AND
||	left to right	logical OR
?:	right to left	conditional
= += -= *= /= %=	right to left	assignment
,	left to right	comma

Figure 2. Operator precedence and associativity

Defining Arrays

Arrays occupy space in memory. You specify the type of each element and the number of elements each array requires so that the computer may reserve the appropriate amount of memory. The following definition reserves 12 elements for integer array c, which has indices in the range 0–11.

When declaring arrays, you need to specify the Name, Type of array, and the Number of elements

```
arrayType arrayName[ numberOfElements ];
```

for examples

```
int c[12];
float myArray[ 3284 ];
```

Declaring multiple arrays of same type

```
int b[100], x[27];
```

reserves 100 elements for integer array b and 27 elements for integer array x. These arrays have indices in the ranges 0–99 and 0–26, respectively. Though you can define multiple arrays at once, defining only one per line is preferred, so you can add a comment explaining each array's purpose. Arrays may contain other data types. For example, an array of type char can store a character string.

Array Examples

This section presents several examples that demonstrate how to define and initialize arrays, and how to perform many common array manipulations.

Defining an Array and Using a Loop to Set the Array's Element Values

Like any other variables, uninitialized array elements contain garbage values. Program 1 uses for statements to set the elements of a five-element integer array n to zeros (lines 11–13) and print the array in tabular format (lines 18–20). The first printf statement (line 15) displays the column heads for the two columns printed in the subsequent for statement.

```
1 // Program_1.c
2 // Initializing the elements of an array to zeros.
3 #include <stdio.h>
4
5 // function main begins program execution
6 int main(void)
7 {
8       int n[5]; // n is an array of five integers
9
10      // set elements of array n to 0
11      for (size_t i = 0; i < 5; ++i) {
12              n[i] = 0; // set element at location i to 0
13      }
14
15      printf("%s%13s\n", "Element", "Value");
16
17      // output contents of array n in tabular format
18      for (size_t i = 0; i < 5; ++i) {
19      printf("%7u%13d\n", i, n[i]);
20      }
21 }
```

output

Element	Value
0	0
1	0
2	0
3	0
4	0

Program 1 Initializing the elements of an array to zeros

Notice that the counter-control variable i is declared to be of type size_t in each for statement (lines 11 and 18), which according to the C standard represents an unsigned integral type.1 This type is recommended for any variable that represents an array's size or indices. Type size_t is defined in header <stddef.h>, which is often included by other headers (such as <stdio.h>).

If you attempt to compile Program 1 and receive errors, simply include <stddef.h> in your program.

On some compilers, size_t represents unsigned int and on others it represents unsigned long. Compilers that use unsigned long typically generate a warning on line 19 of Program 1, because %u is for displaying unsigned ints, not unsigned longs. To eliminate this warning, replace the format specification %u with %lu. Initializing an Array in a Definition with an Initializer List

Initializing an Array in a Definition with an Initializer List

The elements of an array can also be initialized when the array is defined by following the definition with an equals sign and braces, {}, containing a comma-separated list of array initializers. Program 2 initializes an integer array with five values (line 9) and prints the array in tabular format.

```
1 // Program_2.c
2 // Initializing the elements of an array with an initializer list.
3 #include <stdio.h>
4
5 // function main begins program execution
6 int main(void)
7 {
8       // use initializer list to initialize array n
9       int n[5] = {32, 27, 64, 18, 95};
10
11      printf("%s%13s\n", "Element", "Value");
12
13      // output contents of array in tabular format
14      for (size_t i = 0; i < 5; ++i) {
15              printf("%7u%13d\n", i, n[i]);
16      }
17 }
```

Element	Value
0	32
1	27
2	64
3	18
4	95

Program 2 Initializing the elements of an array with an initializer list

If there are fewer initializers than elements in the array, the remaining elements are initialized to zero. For example, the elements of the array n in Program 1 could have been initialized to zero as follows:

```
int n[10] = {0}; // initializes entire array to zeros
```

This explicitly initializes the first element to zero and initializes the remaining nine elements to zero because there are fewer initializers than there are elements in the array. Arrays are not automatically initialized to zero. You must at least initialize the first element to zero for the remaining elements to be automatically zeroed. Array elements are initialized before program startup for static arrays and at runtime for automatic arrays.

Common Programming Error. Forgetting to initialize the elements of an array.
It is a syntax error to provide more initializers in an array initializer list than there are elements in the array—for example, int n[3] = {32, 27, 64, 18}; is a syntax error, because there are four initializers but only three array elements.

If the array size is omitted from a definition with an initializer list, the number of elements in the array will be the number of elements in the initializer list. For example,

```
int n[] = {1, 2, 3, 4, 5};
```

would create a five-element array initialized with the indicated values.

Specifying an Array's Size with a Symbolic Constant and Initializing Array Elements with Calculations

Program 3 initializes the elements of a five-element array s to the values 2, 4, 6, …, 10 and prints the array in tabular format. The values are generated by multiplying the loop counter by 2 and adding 2.

```
1 // Program_3.c
2 // Initializing the elements of array s to the even integers from 2 to 10.
3 #include <stdio.h>
4 #define SIZE 5 // maximum size of array
56
// function main begins program execution
7 int main(void)
8 {
9       // symbolic constant SIZE can be used to specify array size
```

```
10      int s[SIZE]; // array s has SIZE elements
11
12      for (size_t j = 0; ; ++j) { // set the values
13              s[j] = 2 + 2 * j;
14      }
15
16      printf("%s%13s\n", "Element", "Value");
17
18      // output contents of array s in tabular format
19      for (size_t j = 0; ; ++j) {
20              printf("%7u%13d\n", j, s[j]);
21      }
22 }
```

output

Element	Value
0	2
1	4
2	6
3	8
4	10

Program 3 Initializing the elements of array s to the even integers from 2 to 10

The #define preprocessor directive is introduced in this program. Line 4

```
#define SIZE 5
```

defines a symbolic constant SIZE whose value is 5. A symbolic constant is an identifier that's replaced with replacement text by the C preprocessor before the program is compiled. When the program is preprocessed, all occurrences of the symbolic constant SIZE are replaced with the replacement text 5. Using symbolic constants to specify array sizes makes programs more modifiable. In Program 3, we could have the first for loop (line 12) fill a 1000-element array by simply changing the value of SIZE in the #define directive from 5 to 1000. If the symbolic constant SIZE had not been used, we'd have to change the program in lines 10, 12 and 19.

As programs get larger, this technique becomes more useful for writing clear, easy to read, maintainable programs—a symbolic constant (like SIZE) is easier to understand than the numeric value 5, which could have different meanings throughout the code.

Common Programming Error. Ending a #define or #include preprocessor directive with a semicolon. Remember that preprocessor directives are not C statements.

If you terminate the #define preprocessor directive in line 4 with a semicolon, the preprocessor replaces all occurrences of the symbolic constant SIZE in the program with the text "5;". This may lead to syntax errors at compile time, or logic errors at execution time. Remember that the preprocessor is not the C compiler. Defining the size of each array as a symbolic constant makes programs more modifiable.

Common Programming Error. Assigning a value to a symbolic constant in an executable statement is a syntax error. The compiler does not reserve space for symbolic constants as it does for variables that hold values at execution time.

Good Programming Practice. Use only uppercase letters for symbolic constant names. This makes these constants stand out in a program and reminds you that symbolic constants are not variables. In multiword symbolic constant names, separate the words with underscores for readability.

Summing the Elements of an Array

Program 4 sums the values contained in the 12-element integer array a. The for statement's body (line 15) does the totaling.

```
1 // Program_4.c
2 // Computing the sum of the elements of an array.
3 #include <stdio.h>
4 #define SIZE 12
5
6 // function main begins program execution
7 int main(void)
8 {
9       // use an initializer list to initialize the array
10      int a[SIZE] = {1, 3, 5, 4, 7, 2, 99, 16, 45, 67, 89, 45};
11      int total = 0; // sum of array
12
13      // sum contents of array a
14      for (size_t i = 0; i < SIZE; ++i) {
15      total += a[i];
16      }
17
18      printf("Total of array element values is %d\n", total);
19 }
```

output

```
Total of array element values is 383
```

Program 4 Computing the sum of the elements of an array

Using Arrays to Summarize Survey Results

Our next example uses arrays to summarize the results of data collected in a survey. Consider the problem statement. Forty students were asked to rate the quality of the food in the student cafeteria on a scale of 1 to 10 (1 means awful and 10 means excellent). Place the 40 responses in an integer array and summarize the results of the poll.

This is a typical array application (Program 5). We wish to summarize the number of responses of each type (i.e., 1 through 10). The 40-element array responses (lines 14–16) contains the students' responses. We use an 11-element array frequency (line 11) to count the number of occurrences of each response. We ignore frequency[0] because it's logical to have response 1 increment frequency[1] rather than frequency[0]. This allows us to use each response directly as the index in the frequency array.

```
1 // Program_5.c
2 // Analyzing a student poll.
3 #include <stdio.h>
4 #define RESPONSES_SIZE 40 // define array sizes
5 #define FREQUENCY_SIZE 11
6
7 // function main begins program execution
8 int main(void)
9 {
10      // initialize frequency counters to 0
11
12
13      // place the survey responses in the responses array
14      int responses[RESPONSES_SIZE] = {1, 2, 6, 4, 8, 5, 9, 7, 8, 10,
15              1, 6, 3, 8, 6, 10, 3, 8, 2, 7, 6, 5, 7, 6, 8, 6, 7, 5, 6, 6,
16              5, 6, 7, 5, 6, 4, 8, 6, 8, 10};
17
18      // for each answer, select value of an element of array responses
19      // and use that value as an index in array frequency to
20      // determine element to increment
21      for (size_t answer = 0; answer < RESPONSES_SIZE; ++answer) {
22              ++frequency[responses[answer]];
23      }
24
25      // display results
26      printf("%s%17s\n", "Rating", "Frequency");
27
28      // output the frequencies in a tabular format
29      for (size_t rating = 1; rating < FREQUENCY_SIZE; ++rating) {
30      printf("%6d%17d\n", rating, frequency[rating]);
31      }
32 }
```

Rating	Frequency
1	2
2	2
3	2
4	2
5	5
6	11
7	5
8	7
9	1
10	3

Program 5 Analyzing a student poll

Good Programming Practice. Strive for program clarity. Sometimes it may be worthwhile to trade off the most efficient use of memory or processor time in favor of writing clearer programs.

Sometimes performance considerations far outweigh clarity considerations.

The for loop (lines 21–23) takes the responses one at a time from the array responses and increments one of the 10 counters (frequency[1] to frequency[10]) in the frequency array. The key statement in the loop is line 22

```
++frequency[responses[answer]];
```

which increments the appropriate frequency counter depending on the value of the expression responses[answer]. When the counter variable answer is 0, responses[answer] is 1, so ++frequency[responses[answer]]; is interpreted as

```
++frequency[1];
```

which increments array element 1. When answer is 1, the value of responses[answer] is 2, so ++frequency[responses[answer]]; is interpreted as

```
++frequency[2];
```

which increments array element 2. When answer is 2, the value of responses[answer] is 6, so ++frequency[responses[answer]]; is interpreted as

```
++frequency[6];
```

which increments array element 6, and so on. Regardless of the number of responses processed in the survey, only an 11-element array is required (ignoring element zero) to summarize the results. If the data contained invalid values such as 13, the program would attempt to add 1 to frequency[13]. This would be outside the bounds of the array. C has no array bounds checking to prevent the program from referring to an element that does not exist. Thus, an executing program can "walk off" either end of an array without warning— a security problem that we discussed. You should ensure that all array references remain within the bounds of the array.

Common Programming Error. Referring to an element outside the array bounds.

Error-Prevention Tip. Initialize pointers to prevent unexpected results. Programs should validate the correctness of all input values to prevent erroneous information from affecting a program's calculations.

Passing Arrays to Functions

To pass an array argument to a function, specify the array's name without any brackets.

For example, if array hourlyTemperatures has been defined as

int hourlyTemperatures[HOURS_IN_A_DAY];

the function call

modifyArray(hourlyTemperatures, HOURS_IN_A_DAY)

passes array hourlyTemperatures and its size to function modifyArray.

Recall that all arguments in C are passed by value. C automatically passes arrays to functions by reference the called functions can modify the element values in the callers' original arrays. The array's name evaluates to the address of the array's first element. Because the starting address of the array is passed, the called function knows precisely where the array is stored. Therefore, when the called function modifies array elements in its function body, it's modifying the actual elements of the array in their original memory locations.

Program 6 demonstrates that "the value of an array name" is really the address of the first element of the array by printing array, &array[0] and &array using the %p conversion specifier for printing addresses. The %p conversion specifier normally outputs addresses as hexadecimal numbers, but this is compiler dependent. Hexadecimal (base 16) numbers consist of the digits 0 through 9 and the letters A through F (these letters are the hexadecimal equivalents of the decimal numbers 10–15). The output shows that array, &array and &array[0] have the same value, namely 0031F930. The output of this program is system dependent, but the addresses are always identical for a particular execution of this program on a particular computer.

```
1 // Program_6.c
2 // Array name is the same as the address of the array's first element.
3 #include <stdio.h>
4
5 // function main begins program execution
6 int main(void)
7 {
8        char array[5]; // define an array of size 5
9
10       printf(" array = %p\n&array[0] = %p\n &array = %p\n",
11       array, &array[0], &array);
12 }
```


```
array = 0031F930
&array[0] = 0031F930
&array = 0031F930
```

Program 6 Array name is the same as the address of the array's first element

Passing arrays by reference makes sense for performance reasons. If arrays were passed by value, a copy of each element would be passed. For large, frequently passed arrays, this would be time consuming and would consume storage for the copies of the arrays.

Although entire arrays are passed by reference, individual array elements are passed by value exactly as simple variables are. Such simple single pieces of data (such as individual ints, floats and chars) are called scalars. To pass an element of an array to a function, use the indexed name of the array element as an argument in the function call.

For a function to receive an array through a function call, the function's parameter list must specify that an array will be received. For example, the function header for function modifyArray (that we called earlier in this section) might be written as

```
void modifyArray(int b[], size_t size)
```

indicating that modifyArray expects to receive an array of integers in parameter b and the number of array elements in parameter size. The size of the array is not required between the array brackets. If it's included, the compiler checks that it's greater than zero, then ignores it. Specifying a negative size is a compilation error. Because arrays are automatically passed by reference, when the called function uses the array name b, it will be referring to the array in the caller (array hourlyTemperatures in the preceding call).

Difference Between Passing an Entire Array and Passing an Array Element

Program 7 demonstrates the difference between passing an entire array and passing an individual array element. The program first prints the five elements of integer array a (lines 19–21). Next, a and its size are passed to function modifyArray (line 25), where each of a's elements is multiplied by 2 (lines 48–50). Then a is reprinted in main (lines 29–31).

As the output shows, the elements of a are indeed modified by modifyArray. Now the program prints the value of a[3] (line 35) and passes it to function modifyElement (line 37). Function modifyElement multiplies its argument by 2 (line 58) and prints the new value. When a[3] is reprinted in main (line 40), it has not been modified, because individual array elements are passed by value.

```
1 // Program_7.c
2 // Passing arrays and individual array elements to functions.
3 #include <stdio.h>
4 #define SIZE 5
5
6 // function prototypes
7 void modifyArray(int b[], size_t size);
8 void modifyElement(int e);
9
10 // function main begins program execution
11 int main(void)
12 {
13      int a[SIZE] = {0, 1, 2, 3, 4}; // initialize array a
14
15      puts("Effects of passing entire array by reference:\n\nThe "
16           "values of the original array are:");
17
18      // output original array
19      for (size_t i = 0; i < SIZE; ++i) {
20           printf("%3d", a[i]);
21      }
22
23      puts(""); // outputs a newline
24
25      modifyArray(a, SIZE); // pass array a to modifyArray by reference
26      puts("The values of the modified array are:");
27
28      // output modified array
29      for (size_t i = 0; i < SIZE; ++i) {
30           printf("%3d", a[i]);
31      }
32
33      // output value of a[3]
34      printf("\n\n\nEffects of passing array element "
35           "by value:\n\nThe value of a[3] is %d\n", a[3]);
36
37      modifyElement(a[3]); // pass array element a[3] by value
```

```
38
39      // output value of a[3]
40      printf("The value of a[3] is %d\n", a[3]);
41 }
42
43 // in function modifyArray, "b" points to the original array "a"
44 // in memory
45 void modifyArray(int b[], size_t size)
46 {
47      // multiply each array element by 2
48      for (size_t j = 0; j < size; ++j) {
49              b[j] *= 2; // actually modifies original array
50      }
51 }
52
53 // in function modifyElement, "e" is a local copy of array element
54 // a[3] passed from main
55 void modifyElement(int e)
56 {
57      // multiply parameter by 2
58      printf("Value in modifyElement is %d\n", e *= 2);
59 }
```

output

```
Effects of passing entire array by reference:
The values of the original array are:
0 1 2 3 4
The values of the modified array are:
0 2 4 6 8
Effects of passing array element by value:
The value of a[3] is 6
Value in modifyElement is 12
The value of a[3] is 6
```

Program 7 Passing arrays and individual array elements to functions

There may be situations in your programs in which a function should not be allowed to modify array elements. C provides the type qualifier const (for “constant”) that can be used to prevent modification of array values in a function. When an array parameter is preceded by the const qualifier, the array elements become constant in the function body any attempt to modify an element of the array in the function body results in a compile time error.

Sorting Arrays

- ✓ Sorting data can be of utmost of importance. Especially in situations where you need data arranged in a specific order.
- ✓ Sorting an array is the ordering of the array elements in ascending (increasing -from min to max) or descending (decreasing – from max to min) order.

Example:

{2 1 5 3 2} □{1, 2, 2, 3, 5} ascending order
{2 1 5 3 2} □{5, 3, 2, 2, 1} descending order

Program 8 sorts the values in the elements of the 10-element array a (line 10) into ascending order. The technique we use is called the bubble sort or the sinking sort because the smaller values gradually "bubble" their way upward to the top of the array like air bubbles rising in water, while the larger values sink to the bottom of the array. The technique is to make several passes through the array. On each pass, successive pairs of elements (element 0 and element 1, then element 1 and element 2, etc.) are compared. If a pair is in increasing order (or if the values are identical), we leave the values as they are. If a pair is in decreasing order, their values are swapped in the array.

```
1 // Program_8.c
2 // Sorting an array's values into ascending order.
3 #include <stdio.h>
4 #define SIZE 10
5
6 // function main begins program execution
7 int main(void)
8 {
9       // initialize a
10      int a[SIZE] = {2, 6, 4, 8, 10, 12, 89, 68, 45, 37};
11
12      puts("Data items in original order");
13
14      // output original array
15      for (size_t i = 0; i < SIZE; ++i) {
16      printf("%4d", a[i]);
17      }
18
19      // bubble sort
20      // loop to control number of passes
21      for (unsigned int pass = 1; pass < SIZE; ++pass) {
22
23      // loop to control number of comparisons per pass
```

```
24      for (size_t i = 0; i < SIZE - 1; ++i) {
25
26              // compare adjacent elements and swap them if first
27              // element is greater than second element
28                      if (a[i] > a[i + 1]) {
29                              int hold = a[i];
30                              a[i] = a[i + 1];
31                              a[i + 1] = hold;
32                      }
33              }
34      }
35
36      puts("\nData items in ascending order");
37
38      // output sorted array
39      for (size_t i = 0; i < SIZE; ++i) {
40      printf("%4d", a[i]);
41      }
42
43      puts("");
44 }
```


```
Data items in original order
2 6 4 8 10 12 89 68 45 37
Data items in ascending order
2 4 6 8 10 12 37 45 68 89
```

Program 8 Sorting an array's values into ascending order

First the program compares a[0] to a[1], then a[1] to a[2], then a[2] to a[3], and so on until it completes the pass by comparing a[8] to a[9]. Although there are 10 elements, only nine comparisons are performed. Because of the way the successive comparisons are made, a large value may move down the array many positions on a single pass, but a small value may move up only one position. On the first pass, the largest value is guaranteed to sink to the bottom element of the array, a[9]. On the second pass, the second-largest value is guaranteed to sink to a[8]. On the ninth pass, the ninth-largest value sinks to a[1]. This leaves the smallest value in a[0], so only nine passes of the array are needed to sort the array, even though there are ten elements.

The sorting is performed by the nested for loops (lines 21–34). If a swap is necessary, it's performed by the three assignments

```
hold = a[i];
a[i] = a[i + 1];
a[i + 1] = hold;
```

where the extra variable hold temporarily stores one of the two values being swapped. The swap cannot be performed with only the two assignments

```
a[i] = a[i + 1];
```

```
a[i + 1] = a[i];
```

If, for example, a[i] is 7 and a[i + 1] is 5, after the first assignment both values will be 5 and the value 7 will be lost—hence the need for the extra variable hold.

The chief virtue of the bubble sort is that it's easy to program. However, it runs slowly because every exchange moves an element only one position closer to its final destination. This becomes apparent when sorting large arrays. In the exercises, we'll develop more efficient versions of the bubble sort. Far more efficient sorts than the bubble sort have been developed.

Case Study: Computing Mean, Median and Mode Using Arrays

We now consider a larger example. Computers are commonly used for survey data analysis to compile and analyze the results of surveys and opinion polls. Program 9 uses array response initialized with 99 responses to a survey. Each response is a number from 1 to 9. The program computes the mean, median and mode of the 99 values. Program 10 contains a sample run of this program. This example includes most of the common manipulations usually required in array problems, including passing arrays to functions.

```
1 // Program_9.c
2 // Survey data analysis with arrays:
3 // computing the mean, median and mode of the data.
4 #include <stdio.h>
5 #define SIZE 99
6
7 // function prototypes
8 void mean(const unsigned int answer[]);
9 void median(unsigned int answer[]);
10 void mode(unsigned int freq[], unsigned const int answer[]) ;
11 void bubbleSort(int a[]);
12 void printArray(unsigned const int a[]);
13
14 // function main begins program execution
15 int main(void)
16 {
17       unsigned int frequency[10] = {0}; // initialize array frequency
18
19       // initialize array response
20       unsigned int response[SIZE] =
21             {6, 7, 8, 9, 8, 7, 8, 9, 8, 9,
22             7, 8, 9, 5, 9, 8, 7, 8, 7, 8,
23             6, 7, 8, 9, 3, 9, 8, 7, 8, 7,
24             7, 8, 9, 8, 9, 8, 9, 7, 8, 9,
```

```
25              6, 7, 8, 7, 8, 7, 9, 8, 9, 2,
26              7, 8, 9, 8, 9, 8, 9, 7, 5, 3,
27              5, 6, 7, 2, 5, 3, 9, 4, 6, 4,
28              7, 8, 9, 6, 8, 7, 8, 9, 7, 8,
29              7, 4, 4, 2, 5, 3, 8, 7, 5, 6,
30              4, 5, 6, 1, 6, 5, 7, 8, 7};
31
32      // process responses
33      mean(response);
34      median(response);
35      mode(frequency, response);
36 }
37
38 // calculate average of all response values
39 void mean(const unsigned int answer[])
40 {
41      printf("%s\n%s\n%s\n", "********", " Mean", "********");
42
43      unsigned int total = 0; // variable to hold sum of array elements
44
45      // total response values
46      for (size_t j = 0; j < SIZE; ++j) {
47              total += answer[j];
48      }
49
50      printf("The mean is the average value of the data\n"
51              "items. The mean is equal to the total of\n"
52              "all the data items divided by the number\n"
53              "of data items (%u). The mean value for\n"
54              "this run is: %u / %u = %.4f\n\n",
55              SIZE, total, SIZE, (double) total / SIZE);
56 }
57
58 // sort array and determine median element's value
59 void median(unsigned int answer[])
60 {
61      printf("\n%s\n%s\n%s\n%s",
62              "********", " Median", "********",
63              "The unsorted array of responses is");
64
65      printArray(answer); // output unsorted array
66
67      bubbleSort(answer); // sort array
```

```
68
69      printf("%s", "\n\nThe sorted array is");
70      printArray(answer); // output sorted array
71
72      // display median element
73      printf("\n\nThe median is element %u of\n"
74              "the sorted %u element array.\n"
75              "For this run the median is %u\n\n",
76              SIZE / 2, SIZE, );
77      }
78
79 // determine most frequent response
80 void mode(unsigned int freq[], const unsigned int answer[])
81 {
82      printf("\n%s\n%s\n%s\n","********", " Mode", "********");
83
84      // initialize frequencies to 0
85      for (size_t rating = 1; rating <= 9; ++rating) {
86 freq[rating] = 0;
87      }
88
89      // summarize frequencies
90      for (size_t j = 0; j < SIZE; ++j) {
91              ++freq[answer[j]];
92      }
93
94      // output headers for result columns
95      printf("%s%11s%19s\n\n%54s\n%54s\n\n",
96              "Response", "Frequency", "Histogram",
97              "1 1 2 2", "5 0 5 0 5");
98
99      // output results
100     unsigned int largest = 0; // represents largest frequency
101     unsigned int modeValue = 0; // represents most frequent response
102
103     for (rating = 1; rating <= 9; ++rating) {
104             printf("%8u%11u ", rating, freq[rating]);
105
106     // keep track of mode value and largest frequency value
107     if (freq[rating] > largest) {
108             largest = freq[rating];
109             modeValue = rating;
110     }
```

```
111
112     // output histogram bar representing frequency value
113     for (unsigned int h = 1; h <= freq[rating]; ++h) {
114             printf("%s", "*");
115     }
116
117             puts(""); // being new line of output
118     }
119
120     // display the mode value
121     printf("\nThe mode is the most frequent value.\n"
122             "For this run the mode is %u which occurred"
123             " %u times.\n", modeValue, largest);
124 }
125
126 // function that sorts an array with bubble sort algorithm
127 void bubbleSort(unsigned int a[])
128 {
129     // loop to control number of passes
130     for (unsigned int pass = 1; pass < SIZE; ++pass) {
131
132             // loop to control number of comparisons per pass
133             for (size_t j = 0; j < SIZE - 1; ++j) {
134
135                     // swap elements if out of order
136                     if (a[j] > a[j + 1]) {
137                             unsigned int hold = a[j];
138                             a[j] = a[j + 1];
139                             a[j + 1] = hold;
140                     }
141             }
142     }
143 }
144
145 // output array contents (20 values per row)
146 void printArray(const unsigned int a[])
147 {
148     // output array contents
149     for (size_t j = 0; j < SIZE; ++j) {
150
151             if (j % 20 == 0) { // begin new line every 20 values
152                     puts("");
153             }
```

```
154
155             printf("%2u", a[j]);
156     }
157 }
```

output

```
********
Mean
********
The mean is the average value of the data
items. The mean is equal to the total of
all the data items divided by the number
of data items (99). The mean value for
this run is: 681 / 99 = 6.8788

********
Median
********
The unsorted array of responses is
6 7 8 9 8 7 8 9 8 9 7 8 9 5 9 8 7 8 7 8
6 7 8 9 3 9 8 7 8 7 7 8 9 8 9 8 9 7 8 9
6 7 8 7 8 7 9 8 9 2 7 8 9 8 9 8 9 7 5 3
5 6 7 2 5 3 9 4 6 4 7 8 9 6 8 7 8 9 7 8
7 4 4 2 5 3 8 7 5 6 4 5 6 1 6 5 7 8 7
The sorted array is
1 2 2 2 3 3 3 3 4 4 4 4 4 5 5 5 5 5 5 5
5 6 6 6 6 6 6 6 6 6 7 7 7 7 7 7 7 7 7 7
7 7 7 7 7 7 7 7 7 7 7 7 7 8 8 8 8 8 8 8
8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8
9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9
The median is element 49 of
the sorted 99 element array.
For this run the median is 7

********
Mode
********
Response        Frequency       Histogram
                                          1   1   2   2
                                  5   0   5   0   5
        1               1       *
        2               3       ***
        3               4       ****
        4               5       *****
        5               8       ********
        6               9       *********
        7               23      ***********************
        8               27      ***************************
        9               19      *******************
The mode is the most frequent value.
For this run the mode is 8 which occurred 27 times.
```

Program 9 Sample run for the survey data analysis program

Mean. The mean is the arithmetic average of the 99 values. Function mean (Program 9, lines 39–56) computes the mean by totaling the 99 elements and dividing the result by 99.

Median. The median is the middle value. Function median (lines 59–77) determines the median by calling function bubbleSort (defined in lines 127–143) to sort the array of responses into ascending order, then picking answer[SIZE / 2] (the middle element) of the sorted array. When the number of elements is even, the median should be calculated as the mean of the two middle elements. Function median does not currently provide this capability. Function printArray (lines 146–157) is called to output the response array.

Mode. The mode is the value that occurs most frequently among the 99 responses. Function mode (lines 80–124) determines the mode by counting the number of responses of each type, then selecting the value with the greatest count. This version of function mode does not handle a tie Function mode also produces a histogram to aid in determining the mode graphically.

Searching Arrays

You will often work with large amounts of data stored in arrays. It may be necessary to determine whether an array contains a value that matches a certain key value. The process of finding a particular element of an array is called searching. In this section we discuss two searching techniques—the simple linear search technique and the more efficient (but more complex) binary search technique.

Searching an Array with Linear Search

The linear search in Program 10 compares each element of the array with the search key. Because the array is not in any particular order, it's just as likely that the value will be found in the first element as in the last. On average, therefore, the program will have to compare the search key with half the elements of the array.

```
1 // Program_10.c
2 // Linear search of an array.
3 #include <stdio.h>
4 #define SIZE 100
5
6 // function prototype
7
8
9 // function main begins program execution
10 int main(void)
```

```
11 {
12      int a[SIZE]; // create array a
13
14      // create some data
15      for (size_t x = 0; x < SIZE; ++x) {
16              a[x] = 2 * x;
17      }
18
19      printf("Enter integer search key: ");
20      int searchKey; // value to locate in array a
21      scanf("%d", &searchKey);
22
23      // attempt to locate searchKey in array a
24      size_t index = linearSearch(a, searchKey, SIZE);
25
26      // display results
27      if (index != -1) {
28              printf("Found value at index %d\n", index);
29      }
30      else {
31              puts("Value not found");
32      }
33 }
34
35 // compare key to every element of array until the location is found
36 // or until the end of array is reached; return index of element
37 // if key is found or -1 if key is not found
38 size_t linearSearch(const int array[], int key, size_t size)
39 {
40      // loop through array
41      for (size_t n = 0; n < size; ++n) {
42
43              if (array[n] == key) {
44                      return n; // return location of key
45              }
46      }
47
48      return -1; // key not found
49 }
```

Program 10 Linear search of an array.


```
Enter integer search key: 36
Found value at index 18
```

```
Enter integer search key: 37
Value not found
```

Searching an Array with Binary Search

The linear searching method works well for small or unsorted arrays. However, for large arrays linear searching is inefficient. If the array is sorted, the high-speed binary search technique can be used.

The binary search algorithm eliminates from consideration one-half of the elements in a sorted array after each comparison. The algorithm locates the middle element of the array and compares it to the search key. If they're equal, the search key is found and the array index of that element is returned. If they're not equal, the problem is reduced to searching onehalf of the array. If the search key is less than the middle element of the array, the algorithm searches the first half of the array, otherwise the algorithm searches the second half. If the search key is not the middle element in the specified subarray (piece of the original array), the algorithm repeats on one-quarter of the original array. The search continues until the search key is equal to the middle element of a subarray, or until the subarray consists of one element that's not equal to the search key (i.e., the search key is not found).

In a worst case-scenario, searching a sorted array of 1023 elements takes only 10 comparisons using a binary search. Repeatedly dividing 1,024 by 2 yields the values 512, 256, 128, 64, 32, 16, 8, 4, 2 and 1. The number 1,024 (210) is divided by 2 only 10 times to get the value 1. Dividing by 2 is equivalent to one comparison in the binary search algorithm. An array of 1,048,576 (220) elements takes a maximum of only 20 comparisons to find the search key. A sorted array of one billion elements takes a maximum of only 30 comparisons to find the search key. This is a tremendous increase in performance over a linear search of a sorted array, which requires comparing the search key to an average of half of the array elements. For a one-billion-element array, this is a difference between an average of 500 million comparisons and a maximum of 30 comparisons! The maximum comparisons for any array can be determined by finding the first power of 2 greater than the number of array elements.

Program 11 presents the iterative version of function binarySearch (lines 40–68). The function receives four arguments—an integer array b to be searched, an integer searchKey, the low array index and the high array index (these define the portion of the array to be searched). If the search key does not match the middle element of a subarray, the low index or high index is modified so that a smaller subarray can be searched. If the search key is less than the middle element, the high index is set to middle - 1 and the search is continued on the elements from low to middle - 1. If the search key is greater than the middle element, the low index is set to middle + 1 and the search is continued on the elements from middle + 1 to high. The program uses an array of 15 elements. The first power of 2 greater than the number of elements in this array is 16 (24), so no more than 4 comparisons are required to find the search key. The program uses function printHeader (lines 71–88) to output the array indices and function printRow (lines 92–110) to output each subarray during the binary search process. The middle element in each subarray is marked with an asterisk (*) to indicate the element to which the search key is compared.

```
1 // Program_11.c
2 // Binary search of a sorted array.
3 #include <stdio.h>
4 #define SIZE 15
5
6 // function prototypes
7 size_t binarySearch(const int b[], int searchKey, size_t low, size_t high);
8 void printHeader(void);
9 void printRow(const int b[], size_t low, size_t mid, size_t high);
10
11 // function main begins program execution
12 int main(void)
13 {
14       int a[SIZE]; // create array a
15
16       // create data
17       for (size_t i = 0; i < SIZE; ++i) {
18                a[i] = 2 * i;
19       }
20
21       printf("%s", "Enter a number between 0 and 28: ");
22       int key; // value to locate in array a
23       scanf("%d", &key);
24
25       printHeader();
26
27       // search for key in array a
28       size_t result = binarySearch(a, key, 0, SIZE - 1);
29
30       // display results
31       if (result != -1) {
32                printf("\n%d found at index %d\n", key, result);
33       }
34       else {
35       printf("\n%d not found\n", key);
36       }
37 }
38
39 // function to perform binary search of an array
40 size_t binarySearch(const int b[], int searchKey, size_t low, size_t high)
41 {
42       // loop until low index is greater than high index
43       while (low <= high) {
```

```
44
45              // determine middle element of subarray being searched
46              size_t middle = (low + high) / 2;
47
48              // display subarray used in this loop iteration
49              printRow(b, low, middle, high);
50
51              // if searchKey matched middle element, return middle
52              if (searchKey == b[middle]) {
53                      return middle;
54              }
55
56              // if searchKey is less than middle element, set new high
57              else if (searchKey < b[middle]) {
58                      high = middle - 1; // search low end of array
59              } if
60
61              // if searchKey is greater than middle element, set new low
62              else {
63                      low = middle + 1; // search high end of array
64              }
65      } // end while
66
67      return -1; // searchKey not found
68 }
69
70 // Print a header for the output
71 void printHeader(void)
72 {
73      puts("\nIndices:");
74
75      // output column head
76      for (unsigned int i = 0; i < SIZE; ++i) {
77              printf("%3u ", i);
78      }
79
80      puts(""); // start new line of output
81
82      // output line of - characters
83      for (unsigned int i = 1; i <= 4 * SIZE; ++i) {
84              printf("%s", "-");
85      }
86
```

```
87      puts(""); // start new line of output
88 }
89
90 // Print one row of output showing the current
91 // part of the array being processed.
92 void printRow(const int b[], size_t low, size_t mid, size_t high)
93 {
94      // loop through entire array
95      for (size_t i = 0; i < SIZE; ++i) {
96
97              // display spaces if outside current subarray range
98              if (i < low || i > high) {
99                      printf("%s", " ");
100             }
101             else if (i == mid) { // display middle element
102                     printf("%3d*", b[i]); // mark middle value
103             }
104             else { // display other elements in subarray
105                     printf("%3d ", b[i]);
106             }
107     }
108
109     puts(""); // start new line of output
110 }
```

outputs

```
Enter a number between 0 and 28: 25
Indices:
0 1 2 3 4   5   6   7    8    9 10 11   12 13   14
-------------------------------------------------------
0 2 4 6 8 10 12 14* 16 18 20 22   24  26  28
                    16 18 20 22* 24  26  28
                                 24 26* 28
                                 24*
25 not found
```

```
Enter a number between 0 and 28: 8
Indices:
0 1 2 3   4  5     6   7     8   9  10 11 12 13 14
-------------------------------------------------------
0 2 4 6   8 10   12 14* 16 18 20 22 24 26 28
0 2 4 6* 8 10   12
           8 10* 12
           8*
8 found at index 4
```

```
Enter a number between 0 and 28: 6
Indices:
0 1 2 3  4  5 6  7   8   9 10 11 12 13 14
-------------------------------------------------------------
0 2 4 6  8 10 12  14* 16  18 20 22 24 26 28
0 2 4 6* 8 10 12
6 found at index 3
```

Program 11 Binary search of a sorted array

Multidimensional Arrays

Arrays in C can have multiple indices. A common use of multidimensional arrays, which the C standard refers to as multidimensional arrays, is to represent tables of values consisting of information arranged in rows and columns. To identify a particular table element, we must specify two indices: The first (by convention) identifies the element's row and the second (by convention) identifies the element's column. Tables or arrays that require two indices to identify a particular element are called two-dimensional arrays. Multidimensional arrays can have more than two indices.

Figure 1.1 illustrates a two-dimensional array, a. The array contains three rows and four columns, so it's said to be a 3-by-4 array. In general, an array with m rows and n columns is called an m-by-n array.

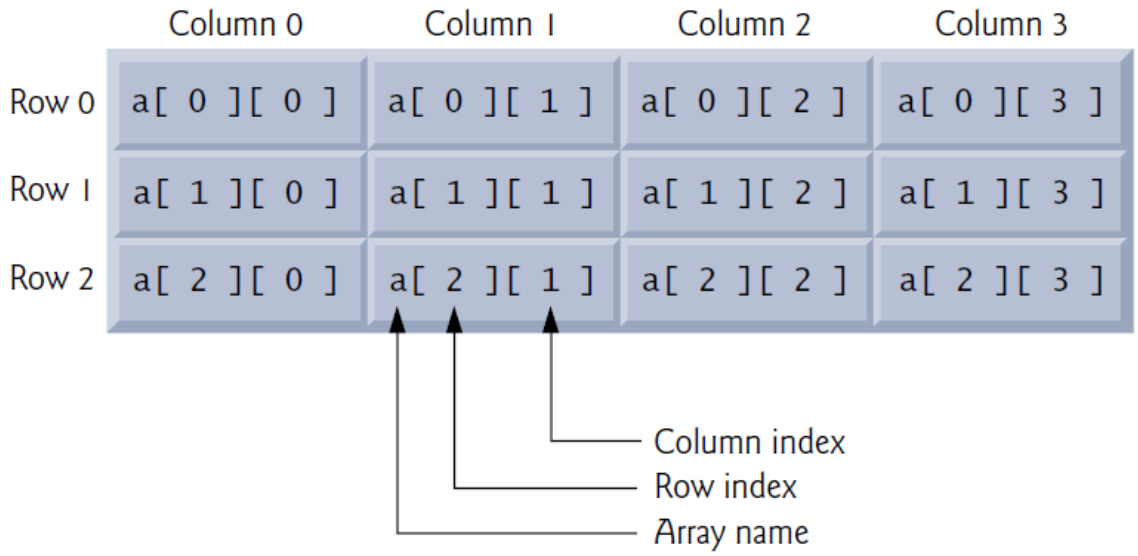

Figure 1.1
Two-dimensional array with three rows and four columns

Every element in array a is identified in Fig. 1.1 by an element name of the form a[i][j]; a is the name of the array, and i and j are the indices that uniquely identify each element in a. The names of the elements in row 0 all have a first index of 0; the names of the elements in column 3 all have a second index of 3.

Referencing a two-dimensional array element as a[x, y] instead of a[x][y] is a logic error. C interprets a[x, y] as a[y] (because the comma in this context is treated as a comma operator), so this programmer error is not a syntax error.

Initializing a Double-Subscripted Array

A multidimensional array can be initialized when it's defined. For example, a two-dimensional array int b[2][2] could be defined and initialized with:

int b[2][2] = {{1, 2}, {3, 4}};

1	2
3	4

The values are grouped by row in braces. The values in the first set of braces initialize row 0 and the values in the second set of braces initialize row 1. So, the values 1 and 2 initialize elements b[0][0] and b[0][1], respectively, and the values 3 and 4 initialize elements b[1][0] and b[1][1], respectively. If there are not enough initializers for a given row, the remaining elements of that row are initialized to 0. Thus:

int b[2][2] = {{1}, {3, 4}};

1	0
3	4

would initialize b[0][0] to 1, b[0][1] to 0, b[1][0] to 3 and b[1][1] to 4.

Referencing its elements, you need to specify its row, and then its column.

```
printf( "%d", b[ 0 ][ 1 ] );
```

Example declaration

int d[2][4];

- ✓ This is an array with two elements, each element is an array of four int values. The elements are laid out sequentially in memory, just like a one-dimensional array. Row order of the elements in the rightmost subscript are stored contiguously.

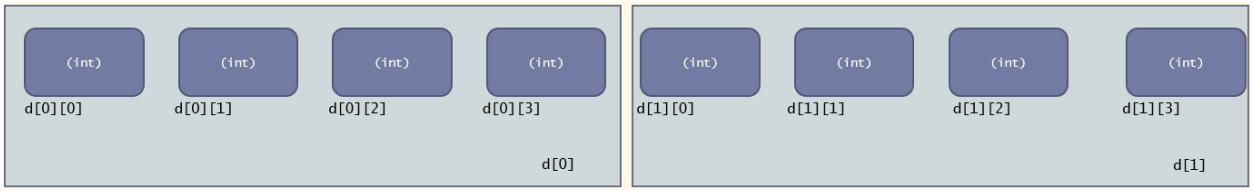

Figure 1.2
Array with two elements

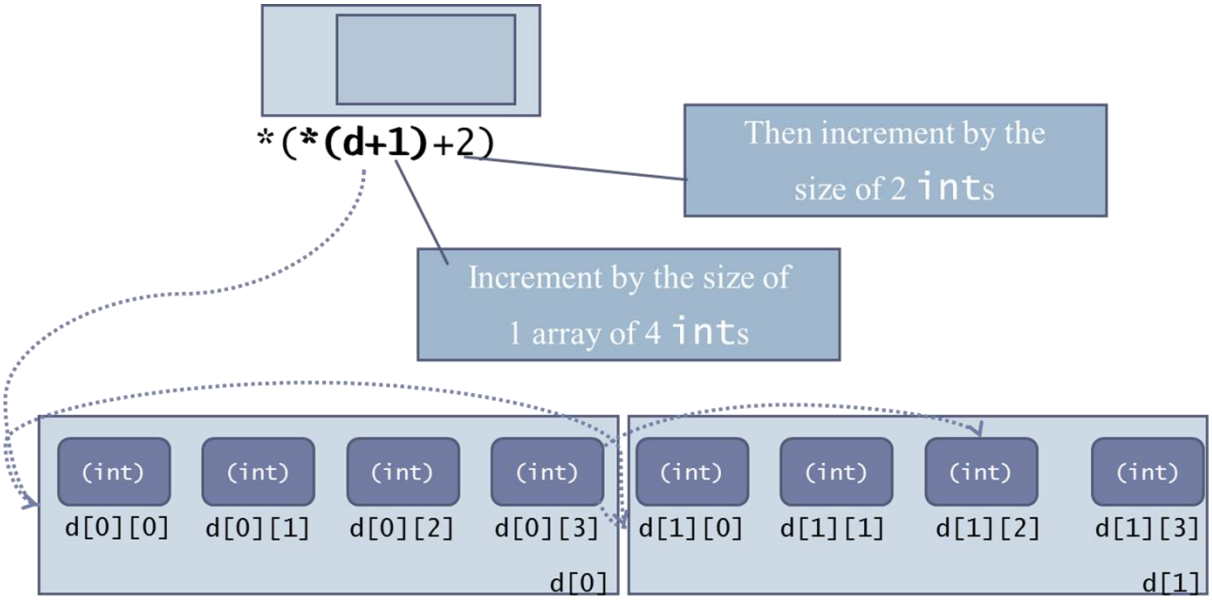

Figure 1.3
Array with two elements

If you keep within the "paradigm" of the multidimensional array, the order do not matter, but if you use tricks with pointer arithmetic, then it matters a lot.

For example

0	1	2	3
4	5	6	7

✓ int d[2][4] = {0, 1, 2, 3, 4, 5, 6, 7};

✗ int d[2][4] = {0, 4, 1, 5, 2, 6, 3, 7};

Program 12 demonstrates defining and initializing two-dimensional arrays.

```
1 // Program_12.c
2 // Initializing multidimensional arrays.
3 #include <stdio.h>
4
5 void printArray(int a[][3]); // function prototype
6
7 // function main begins program execution
8 int main(void)
9 {
10      int array1[2][3] = {{1, 2, 3}, {4, 5, 6}};
11      puts("Values in array1 by row are:");
12      printArray(array1);
13
14      int array2[2][3] = {1, 2, 3, 4, 5};
15      puts("Values in array2 by row are:");
16      printArray(array2);
17
18
19      puts("Values in array3 by row are:");
20      printArray(array3);
21 }
22
23 // function to output array with two rows and three columns
24 void printArray( )
25 {
26      // loop through rows
27      for (size_t i = 0; i <= 1; ++i) {
28
29              // output column values
30              for (size_t j = 0; j <= 2; ++j) {
31                      printf("%d ", );
32              }
33
34              printf("\n"); // start new line of output
35      }
36 }
```

output

```
Values in array1 by row are:
1 2 3
4 5 6
Values in array2 by row are:
1 2 3
4 5 0
Values in array3 by row are:
1 2 0
4 0 0
```

Program 13 Initializing multidimensional arrays

array1 Definition

The program defines three arrays of two rows and three columns (six elements each). The definition of array1 (line 10) provides six initializers in two sublists. The first sublist initializes row 0 of the array to the values 1, 2 and 3; and the second sublist initializes row 1 of the array to the values 4, 5 and 6.

array2 Definition

If the braces around each sublist are removed from the array1 initializer list, the compiler initializes the elements of the first row followed by the elements of the second row. The definition of array2 (line 14) provides five initializers. The initializers are assigned to the first row, then the second row. Any elements that do not have an explicit initializer are initialized to zero automatically, so array2[1][2] is initialized to 0.

array3 Definition

The definition of array3 (line 18) provides three initializers in two sublists. The sublist for the first row explicitly initializes the first two elements of the first row to 1 and 2. The third element is initialized to zero. The sublist for the second row explicitly initializes the first element to 4. The last two elements are initialized to zero.

printArray Function

The program calls printArray (lines 24–36) to output each array's elements. The function definition specifies the array parameter as int a[][3]. In a one-dimensional array parameter, the array brackets are empty. The first index of a multidimensional array is not required, but all subsequent indices are required. The compiler uses these indices to determine the locations in memory of elements in multidimensional arrays. All array elements are stored consecutively in memory regardless of the number of indices. In a two-dimensional array, the first row is stored in memory followed by the second row.

Providing the index values in a parameter declaration enables the compiler to tell the function how to locate an element in the array. In a two-dimensional array, each row is basically a one-dimensional array. To locate an element in a particular row, the compiler must know how many elements are in each row so that it can skip the proper number of memory locations when accessing the array. Thus, when accessing a[1][2] in our example, the compiler knows to skip the three elements of the first row to get to the second row (row 1). Then, the compiler accesses element 2 of that row.

Setting the Elements in One Row

Many common array manipulations use for iteration statements. For example, the following statement sets all the elements in row 2 of array a in figure 1.1 to zero:

```
for (column = 0; column <= 3; ++column) {
        a[2][column] = 0;
}
```

We specified row 2, so the first index is always 2. The loop varies only the second (column) index. The preceding for statement is equivalent to the assignment statements:

```
a[2][0] = 0;
a[2][1] = 0;
a[2][2] = 0;
a[2][3] = 0;
```

Totaling the Elements in a Two-Dimensional Array

The following nested for statement determines the total of all the elements in array a.

```
total = 0;
for (row = 0; row <= 2; ++row) {
        for (column = 0; column <= 3; ++column) {
                total += a[row][column];
        }
}
```

The for statement totals the elements of the array one row at a time. The outer for statement begins by setting row (i.e., the row index) to 0 so that the elements of that row may be totaled by the inner for statement. The outer for statement then increments row to 1, so the elements of that row can be totaled. Then, the outer for statement increments row to 2, so the elements of the third row can be totaled. When the nested for statement terminates, total contains the sum of all the elements in the array a.

Two-Dimensonal Array Manipulations

Program 13 performs several other common array manipulations on a 3-by-4 array studentGrades using for statements. Each row of the array represents a student and each column represents a grade on one of the four exams the students took during the semester. The array manipulations are performed by four functions. Function minimum (lines 39– 56) determines the lowest grade of any student for the semester. Function maximum (lines 59–76) determines the highest grade of any student for the semester. Function average (lines 79–89) determines a particular student's semester average. Function printArray (lines 92–108) outputs the two-dimensional array in a neat, tabular format.

```
1 // Program_13.c
2 // Two-dimensional array manipulations.
3 #include <stdio.h>
4 #define STUDENTS 3
5 #define EXAMS 4
6
7 // function prototypes
8 int minimum(const int grades[][EXAMS], size_t pupils, size_t tests);
9 int maximum(const int grades[][EXAMS], size_t pupils, size_t tests);
10 double average(const int setOfGrades[], size_t tests);
11 void printArray(const int grades[][EXAMS], size_t pupils, size_t tests);
12
13 // function main begins program execution
14 int main(void)
15 {
16       // initialize student grades for three students (rows)
17       int studentGrades[STUDENTS][EXAMS] =
18                 { { 77, 68, 86, 73 },
19                           { 96, 87, 89, 78 },
20                           { 70, 90, 86, 81 } };
21
22       // output array studentGrades
23       puts("The array is:");
24       printArray(studentGrades, STUDENTS, EXAMS);
25
26       // determine smallest and largest grade values
27       printf("\n\nLowest grade: %d\nHighest grade: %d\n",
28                 minimum(studentGrades, STUDENTS, EXAMS),
29                 maximum(studentGrades, STUDENTS, EXAMS));
30
31       // calculate average grade for each student
32       for (size_t student = 0; student < STUDENTS; ++student) {
33                 printf("The average grade for student %u is %.2f\n",
34                           student, );
35       }
36 }
37
38 // Find the minimum grade
39 int minimum(const int grades[][EXAMS], size_t pupils, size_t tests)
40 {
41       int lowGrade = 100; // initialize to highest possible grade
42
43       // loop through rows of grades
```

```
44      for (size_t i = 0; i < pupils; ++i) {
45
46              // loop through columns of grades
47              for (size_t j = 0; j < tests; ++j) {
48
49                      if (grades[i][j] < lowGrade) {
50                              lowGrade = grades[i][j];
51                      }
52              }
53      }
54
55      return lowGrade; // return minimum grade
56 }
57
58 // Find the maximum grade
59 int maximum(const int grades[][EXAMS], size_t pupils, size_t tests)
60 {
61      int highGrade = 0; // initialize to lowest possible grade
62
63      // loop through rows of grades
64      for (size_t i = 0; i < pupils; ++i) {
65
66              // loop through columns of grades
67              for (size_t j = 0; j < tests; ++j) {
68
69                      if (grades[i][j] > highGrade) {
70                              highGrade = grades[i][j];
71                      }
72              }
73      }
74
75      return highGrade; // return maximum grade
76 }
77
78 // Determine the average grade for a particular student
79 double average(const int setOfGrades[], size_t tests)
80 {
81      int total = 0; // sum of test grades
82
83      // total all grades for one student
84      for (size_t i = 0; i < tests; ++i) {
85              total += setOfGrades[i];
86 }
```

```
87
88      return (double) total / tests; // average
89 }
90
91 // Print the array
92 void printArray(const int grades[][EXAMS], size_t pupils, size_t tests)
93 {
94      // output column heads
95      printf("%s", " [0] [1] [2] [3]");
96
97      // output grades in tabular format
98      for (size_t i = 0; i < pupils; ++i) {
99
100             // output label for row
101             printf("\nstudentGrades[%u] ", i);
102
103             // output grades for one student
104             for (size_t j = 0; j < tests; ++j) {
105                     printf("%-5d", grades[i][j]);
106             }
107     }
108 }
```

output

```
The array is:
                    [0]  [1]  [2] [3]
studentGrades [0]    77   68   86  73
studentGrades [1]    96   87   89  78
studentGrades [2]    70   90   86  81

Lowest grade: 68
Highest grade: 96
The average grade for student 0 is 76.00
The average grade for student 1 is 87.50
The average grade for student 2 is 81.75
```

Program 13 Two-dimensional array manipulations

Functions minimum, maximum and printArray each receive three arguments—the studentGrades array (called grades in each function), the number of students (rows of the array) and the number of exams (columns of the array). Each function loops through array grades using nested for statements. The following nested for statement is from the function minimum definition:

```
// loop through rows of grades
for (i = 0; i < pupils; ++i) {
        // loop through columns of grades
        for (j = 0; j < tests; ++j) {
                if (grades[i][j] < lowGrade) {
                        lowGrade = grades[i][j];
                }
        }
}
```

The outer for statement begins by setting i (i.e., the row index) to 0 so the elements of that row (i.e., the grades of the first student) can be compared to variable lowGrade in the body of the inner for statement. The inner for statement loops through the four grades of a particular row and compares each grade to lowGrade. If a grade is less than lowGrade, lowGrade is set to that grade. The outer for statement then increments the row index to 1. The elements of that row are compared to variable lowGrade. The outer for statement then increments the row index to 2. The elements of that row are compared to variable lowGrade. When execution of the nested statement is complete, lowGrade contains the smallest grade in the two-dimensional array. Function maximum works similarly to function minimum.

Function average (lines 79–89) takes two arguments—a two-dimensional array of test results for a particular student called setOfGrades and the number of test results in the array. When average is called, the first argument studentGrades[student] is passed. This causes the address of one row of the two-dimensional array to be passed to average. The argument studentGrades[1] is the starting address of row 1 of the array. Remember that a two-dimensional array is basically an array of one-dimensional arrays and that the name of a one-dimensional array is the address of the array in memory. Function average calculates the sum of the array elements, divides the total by the number of test results and returns the floating-point result.

References

Paul Deitel, and Harvey Deitel, “C How to Program with an introduction to C++” Eight Edition

Kimberly Nelson King, “C Programming: A Modern Approach” Second Edition

Brian W. Kernighan and Dennis M. Ritchie, “C Programming Language” 2nd Edition

https://www.thecrazyprogrammer.com

https://www.geeksforgeeks.org

Additional Resources and Links

Definition of Array by Neso Academy https://www.youtube.com/watch?v=55l-aZ7_F24&list=PLBlnK6fEyqRhX6r2uhhlubuF5QextdCSM&index=86

Declaration of Array by Neso Academy https://www.youtube.com /watch?v= Bqud0_ozgcc&list=PLBlnK6fEyqRhX6r2uhhlubuF5QextdCSM&index=87

Initializing an Array by Neso Academy https://www.youtube.com /watch?v=GtcRReso9Xk&list=PLBlnK6fEyqRhX6r2uhhlubuF5QextdCSM&index=89

Designated Initializion of Arrays by Neso Academy https://www.youtube.com/ watch?v=Sr21AdNJPKg&list=PLBlnK6fEyqRhX6r2uhhlubuF5QextdCSM&index=90

Arrays in C(solved problem 1) by neso Academy https://www.youtube.com/ watch?v=hsmSDBBsifo&list=PLBlnK6fEyqRhX6r2uhhlubuF5QextdCSM&index=91

Arrays in C(solved problem 2) by Neso Academy https://www.youtube.com /watch?v=C57wwOOF6ys&list=PLBlnK6fEyqRhX6r2uhhlubuF5QextdCSM&index=92

Introduction to Multi-Dimensional Arrays by Neso Academy https://www.youtube.com/watch?v=36z4qgN3GWw&list=PLBlnK6fEyqRhX6r2uhhlubuF5QextdCSM&index=94

Introduction to Multi-Dimensional Arrays (2D) by Neso Academy https://www.youtube.com/watch?v=36z4qgN3GWw&list=PLBlnK6fEyqRhX6r2uhhlubuF5QextdCSM&index=95

Introduction to Multi-Dimensional Arrays (2D) by Neso Academy https://www.youtube.com/watch?v=bbkdiUbou74&list=PLBlnK6fEyqRhX6r2uhhlubuF5QextdCSM&index=96

Multi-Dimensional Arrays by Neso Academy
https://www.youtube.com/watch?v=36z4qgN3GWw

C Program for Matrix Multiplication (Part 1) by Neso Academy https://www.youtube.com/watch?v=aAFP5wsmH2k&list=PLBlnK6fEyqRhX6r2uhhlubuF5QextdCSM&index=98

C Program for Matrix Multiplication (Part 2) by Neso Academy https://www.youtube.com/watch?v=jzdQqoG1tZs&list=PLBlnK6fEyqRhX6r2uhhlubuF5QextdCSM&index=99

Chapter Assessment

I. Answer each of the following:

a. Lists and tables of values are stored in?
b. The number used to refer to a particular element of an array is called?
c. A(n) _____ should be used to specify the size of an array because it makes the program more modifiable.
d. The process of placing the elements of an array in order is called _____ the array.
e. Determining whether an array contains a certain key value is called _____ the array.
f. An array that uses two indices is referred to as a(n) _____ array.

II. State whether the following are true or false. If the answer is false, explain why.

a. An array can store many different types of values.
b. An array index can be of data type double.
c. If there are fewer initializers in an initializer list than the number of elements in the array, C automatically initializes the remaining elements to the last value in the list of initializers.
d. It's an error if an initializer list contains more initializers than there are array elements.
e. An individual array element that's passed to a function as an argument of the form a[i] and modified in the called function will contain the modified value in the calling function.

III. Follow the instructions below regarding an array called fractions.

a. Define a symbolic constant SIZE to be replaced with the replacement text 10.
b. Define an array with SIZE elements of type double and initialize the elements to 0.
c. Refer to array element 4.
d. Assign the value 1.667 to array element nine.
e. Assign the value 3.333 to the seventh element of the array.
f. Print array elements 6 and 9 with two digits of precision to the right of the decimal
g. point, and show the output that's displayed on the screen.
h. Print all the elements of the array, using a for iteration statement. Assume the integer
i. variable x has been defined as a control variable for the loop. Show the output.

IV. Write statements to accomplish the following:

a. Define table to be an integer array and to have 3 rows and 3 columns. Assume the symbolic constant SIZE has been defined to be 3.
b. How many elements does the array table contain? Print the total number of elements.
c. Use a for iteration statement to initialize each element of table to the sum of its indices. Assume the integer variables x and y are defined as control variables.

d. Print the values of each element of array table. Assume the array was initialized with the definition:

```
int table[SIZE][SIZE] =
{ { 1, 8 }, { 2, 4, 6 }, { 5 } };
```

V. Find the error in each of the following program segments and correct the error.

a. #define SIZE 100;
b. SIZE = 10;
c. Assume int b[10] = { 0 }, i;
```
for (i = 0; i <= 10; ++i) {
        b[i] = 1;
}
```
d. #include <stdio.h>;
e. Assume int a[2][2] = { { 1, 2 }, { 3, 4 } };
```
a[1, 1] = 5;
```
f. #define VALUE = 120

VI. Write the following programs:

a. that fills the left-to-right diagonal of a square matrix (a 2D array with an equal number of rows and columns) with zeros, the lower left triangle with –1s, and the upper right triangle with +1s. The output of the program, assuming a six-by-six matrix, is

0	1	1	1	1	1
-1	0	1	1	1	1
-1	-1	0	1	1	1
-1	-1	-1	0	1	1
-1	-1	-1	-1	0	1
-1	-1	-1	-1	-1	0

1. Initialize variables
   1.1 Define functions to take double scripted arrays
   1.2 Initialize studentgrades[][]
2. Call functions minimum, maximum, and average

b. Write a program that will get 10 integer values from the user and stores them into an array named list, then output the contents of the array on the screen. After displaying the values from the array, the program should prompt the user to enter a key value that will be used in as key the array. Your program should output on the screen the values in the array that are less than the key and the values in the array that are greater than the key. Display also an appropriate message notifying the user whether the key value is found in the array with its corresponding position. Implement as functions with array as parameter some of the tasks specified above.

Notes

Notes

Notes