02_06_C Arrays

Objectives

In this chapter, you'll:

- Use the array data structure to represent lists and tables of values.
- Define an array, initialize an array and refer to individual elements of an array.
- Define symbolic constants.
- Pass arrays to functions.
- Use arrays to store, sort and search lists and tables of values.
- Define and manipulate multidimensional arrays.
- Create variable-length arrays whose size is determined at execution time.
- Understand security issues related to input with `scanf`, output with `printf` and arrays.

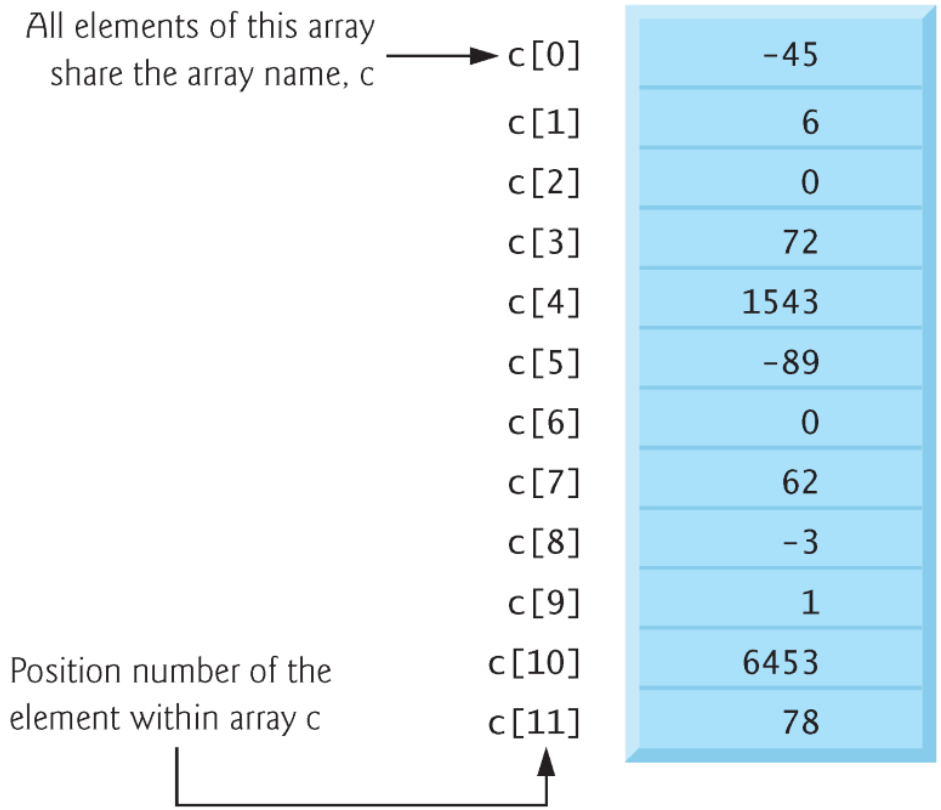

Fig. 6.1 | 12-element array.

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

Fig. 6.2 | Operator precedence and associativity.

```c
// Fig. 6.3: fig06_03.c
// Initializing the elements of an array to zeros.
#include <stdio.h>

// function main begins program execution
int main(void)
{
   int n[5]; // n is an array of five integers

   // set elements of array n to 0
   for (size_t i = 0; i < 5; ++i) {
      n[i] = 0; // set element at location i to 0
   }

   printf("%s%13s\n", "Element", "Value");

   // output contents of array n in tabular format
   for (size_t i = 0; i < 5; ++i) {
      printf("%7u%13d\n", i, n[i]);
   }
}
```

Fig. 6.3 | Initializing the elements of an array to zeros. (Part 1 of 2.)

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

Fig. 6.3 | Initializing the elements of an array to zeros. (Part 2 of 2.)

```c
// Fig. 6.4: fig06_04.c
// Initializing the elements of an array with an initializer list.
#include <stdio.h>

// function main begins program execution
int main(void)
{
   // use initializer list to initialize array n
   int n[5] = {32, 27, 64, 18, 95};

   printf("%s%13s\n", "Element", "Value");

   // output contents of array in tabular format
   for (size_t i = 0; i < 5; ++i) {
      printf("%7u%13d\n", i, n[i]);
   }
}
```

Fig. 6.4 | Initializing the elements of an array with an initializer list. (Part 1 of 2.)

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

Fig. 6.4 | Initializing the elements of an array with an initializer list. (Part 2 of 2.)

```c
// Fig. 6.5: fig06_05.c
// Initializing the elements of array s to the even integers from 2 to 10.
#include <stdio.h>
#define SIZE 5 // maximum size of array

// function main begins program execution
int main(void)
{
   // symbolic constant SIZE can be used to specify array size
   int s[SIZE]; // array s has SIZE elements

   for (size_t j = 0; j < SIZE; ++j) { // set the values
      s[j] = 2 + 2 * j;
   }

   printf("%s%13s\n", "Element", "Value");

   // output contents of array s in tabular format
   for (size_t j = 0; j < SIZE; ++j) {
      printf("%7u%13d\n", j, s[j]);
   }
}
```

Fig. 6.5 | Initializing the elements of array `s` to the even integers from 2 to 10. (Part 1 of 2.)

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

Fig. 6.5 | Initializing the elements of array `s` to the even integers from 2 to 10. (Part 2 of 2.)

```c
// Fig. 6.6: fig06_06.c
// Computing the sum of the elements of an array.
#include <stdio.h>
#define SIZE 12

// function main begins program execution
int main(void)
{
   // use an initializer list to initialize the array
   int a[SIZE] = {1, 3, 5, 4, 7, 2, 99, 16, 45, 67, 89, 45};
   int total = 0; // sum of array

   // sum contents of array a
   for (size_t i = 0; i < SIZE; ++i) {
      total += a[i];
   }

   printf("Total of array element values is %d\n", total);
}
```

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

Fig. 6.6 | Computing the sum of the elements of an array.

```c
// Fig. 6.7: fig06_07.c
// Analyzing a student poll.
#include <stdio.h>
#define RESPONSES_SIZE 40 // define array sizes
#define FREQUENCY_SIZE 11

// function main begins program execution
int main(void)
{
   // initialize frequency counters to 0
   int frequency[FREQUENCY_SIZE] = {0};

   // place the survey responses in the responses array
   int responses[RESPONSES_SIZE] = {1, 2, 6, 4, 8, 5, 9, 7, 8, 10,
        1, 6, 3, 8, 6, 10, 3, 8, 2, 7, 6, 5, 7, 6, 8, 6, 7, 5, 6, 6,
        5, 6, 7, 5, 6, 4, 8, 6, 8, 10};

   // for each answer, select value of an element of array responses
   // and use that value as an index in array frequency to
   // determine element to increment
   for (size_t answer = 0; answer < RESPONSES_SIZE; ++answer) {
      ++frequency[responses[answer]];
   }

```

Fig. 6.7 | Analyzing a student poll. (Part 1 of 2.)

```
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
```

Fig. 6.7 | Analyzing a student poll. (Part 2 of 2.)

```c
// Fig. 6.8: fig06_08.c
// Displaying a histogram.
#include <stdio.h>
#define SIZE 5

// function main begins program execution
int main(void)
{
   // use initializer list to initialize array n
   int n[SIZE] = {19, 3, 15, 7, 11};

   printf("%s%13s%17s\n", "Element", "Value", "Histogram");

   // for each element of array n, output a bar of the histogram
   for (size_t i = 0; i < SIZE; ++i) {
      printf("%7u%13d          ", i, n[i]);

      for (int j = 1; j <= n[i]; ++j) { // print one bar
         printf("%c", '*');
      }

      puts(""); // end a histogram bar with a newline
   }
}
```

Fig. 6.8 | Displaying a histogram. (Part 1 of 2.)

```
Element         Value      Histogram
      0            19      ********************
      1             3      ***
      2            15      ***************
      3             7      *******
      4            11      ***********
```

Fig. 6.8 | Displaying a histogram. (Part 2 of 2.)

```c
// Fig. 6.9: fig06_09.c
// Roll a six-sided die 60,000,000 times
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define SIZE 7

// function main begins program execution
int main(void)
{
   unsigned int frequency[SIZE] = {0}; // clear counts

   srand(time(NULL)); // seed random number generator

   // roll die 60,000,000 times
   for (unsigned int roll = 1; roll <= 60000000; ++roll) {
      size_t face = 1 + rand() % 6;
      ++frequency[face]; // replaces entire switch of Fig. 5.12
   }

```

Fig. 6.9 | Roll a six-sided die 60,000,000 times. (Part 1 of 2.)

```
21      printf("%s%17s\n", "Face", "Frequency");
22
23      // output frequency elements 1-6 in tabular format
24      for (size_t face = 1; face < SIZE; ++face) {
25         printf("%4d%17d\n", face, frequency[face]);
26      }
27   }
```

```
Face        Frequency
   1          9997167
   2         10003506
   3         10001940
   4          9995833
   5         10000843
   6         10000711
```

Fig. 6.9 | Roll a six-sided die 60,000,000 times. (Part 2 of 2.)

```c
// Fig. 6.10: fig06_10.c
// Treating character arrays as strings.
#include <stdio.h>
#define SIZE 20

// function main begins program execution
int main(void)
{
   char string1[SIZE]; // reserves 20 characters
   char string2[] = "string literal"; // reserves 15 characters

   // read string from user into array string1
   printf("%s", "Enter a string (no longer than 19 characters): ");
   scanf("%19s", string1); // input no more than 19 characters

   // output strings
   printf("string1 is: %s\nstring2 is: %s\n"
          "string1 with spaces between characters is:\n",
          string1, string2);
```

Fig. 6.10 | Treating character arrays as strings. (Part 1 of 2.)

```
20
21      // output characters until null character is reached
22      for (size_t i = 0; i < SIZE && string1[i] != '\0'; ++i) {
23         printf("%c ", string1[i]);
24      }
25
26      puts("");
27   }
```

```
Enter a string (no longer than 19 characters): Hello there
string1 is: Hello
string2 is: string literal
string1 with spaces between characters is:
H e l l o
```

Fig. 6.10 | Treating character arrays as strings. (Part 2 of 2.)

```c
// Fig. 6.11: fig06_11.c
// Static arrays are initialized to zero if not explicitly initialized.
#include <stdio.h>

void staticArrayInit(void); // function prototype
void automaticArrayInit(void); // function prototype

// function main begins program execution
int main(void)
{
   puts("First call to each function:");
   staticArrayInit();
   automaticArrayInit();

   puts("\n\nSecond call to each function:");
   staticArrayInit();
   automaticArrayInit();
}

```

Fig. 6.11 | Static arrays are initialized to zero if not explicitly initialized. (Part 1 of 4.)

```
20  // function to demonstrate a static local array
21  void staticArrayInit(void)
22  {
23     // initializes elements to 0 before the function is called
24     static int array1[3];
25
26     puts("\nValues on entering staticArrayInit:");
27
28     // output contents of array1
29     for (size_t i = 0; i <= 2; ++i) {
30        printf("array1[%u] = %d  ", i, array1[i]);
31     }
32
33     puts("\nValues on exiting staticArrayInit:");
34
35     // modify and output contents of array1
36     for (size_t i = 0; i <= 2; ++i) {
37        printf("array1[%u] = %d  ", i, array1[i] += 5);
38     }
39  }
40
```

Fig. 6.11 | Static arrays are initialized to zero if not explicitly initialized. (Part 2 of 4.)

```
41  // function to demonstrate an automatic local array
42  void automaticArrayInit(void)
43  {
44     // initializes elements each time function is called
45     int array2[3] = {1, 2, 3};
46
47     puts("\n\nValues on entering automaticArrayInit:");
48
49     // output contents of array2
50     for (size_t i = 0; i <= 2; ++i) {
51        printf("array2[%u] = %d  ", i, array2[i]);
52     }
53
54     puts("\nValues on exiting automaticArrayInit:");
55
56     // modify and output contents of array2
57     for (size_t i = 0; i <= 2; ++i) {
58        printf("array2[%u] = %d  ", i, array2[i] += 5);
59     }
60  }
```

Fig. 6.11 | Static arrays are initialized to zero if not explicitly initialized. (Part 3 of 4.)

```
First call to each function:

Values on entering staticArrayInit:
array1[0] = 0  array1[1] = 0  array1[2] = 0
Values on exiting staticArrayInit:
array1[0] = 5  array1[1] = 5  array1[2] = 5

Values on entering automaticArrayInit:
array2[0] = 1  array2[1] = 2  array2[2] = 3
Values on exiting automaticArrayInit:
array2[0] = 6  array2[1] = 7  array2[2] = 8

Second call to each function:

Values on entering staticArrayInit:
array1[0] = 5  array1[1] = 5  array1[2] = 5 —— values preserved from last call
Values on exiting staticArrayInit:
array1[0] = 10  array1[1] = 10  array1[2] = 10

Values on entering automaticArrayInit:
array2[0] = 1  array2[1] = 2  array2[2] = 3 —— values reinitialized after last call
Values on exiting automaticArrayInit:
array2[0] = 6  array2[1] = 7  array2[2] = 8
```

Fig. 6.11 | Static arrays are initialized to zero if not explicitly initialized. (Part 4 of 4.)

```c
// Fig. 6.12: fig06_12.c
// Array name is the same as the address of the array's first element.
#include <stdio.h>

// function main begins program execution
int main(void)
{
   char array[5]; // define an array of size 5

   printf("    array = %p\n&array[0] = %p\n   &array = %p\n",
      array, &array[0], &array);
}
```

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

Fig. 6.12 | Array name is the same as the address of the array's first element.

```c
// Fig. 6.13: fig06_13.c
// Passing arrays and individual array elements to functions.
#include <stdio.h>
#define SIZE 5

// function prototypes
void modifyArray(int b[], size_t size);
void modifyElement(int e);

// function main begins program execution
int main(void)
{
   int a[SIZE] = {0, 1, 2, 3, 4}; // initialize array a

   puts("Effects of passing entire array by reference:\n\nThe "
      "values of the original array are:");

   // output original array
   for (size_t i = 0; i < SIZE; ++i) {
      printf("%3d", a[i]);
   }

   puts(""); // outputs a newline

```

Fig. 6.13 | Passing arrays and individual array elements to functions. (Part 1 of 4.)

```
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
```

Fig. 6.13 | Passing arrays and individual array elements to functions. (Part 2 of 4.)

```
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  }
```

Fig. 6.13 | Passing arrays and individual array elements to functions. (Part 3 of 4.)

```
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
```

Fig. 6.13 | Passing arrays and individual array elements to functions. (Part 4 of 4.)

```
1  // in function tryToModifyArray, array b is const, so it cannot be
2  // used to modify its array argument in the caller
3  void tryToModifyArray(const int b[])
4  {
5     b[0] /= 2; // error
6     b[1] /= 2; // error
7     b[2] /= 2; // error
8  }
```

Fig. 6.14 | Using the `const` type qualifier with arrays.

```c
// Fig. 6.15: fig06_15.c
// Sorting an array's values into ascending order.
#include <stdio.h>
#define SIZE 10

// function main begins program execution
int main(void)
{
   // initialize a
   int a[SIZE] = {2, 6, 4, 8, 10, 12, 89, 68, 45, 37};

   puts("Data items in original order");

   // output original array
   for (size_t i = 0; i < SIZE; ++i) {
      printf("%4d", a[i]);
   }

```

Fig. 6.15 | Sorting an array's values into ascending order. (Part 1 of 3.)

```
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
```

Fig. 6.15 | Sorting an array's values into ascending order. (Part 2 of 3.)

```
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
```

Fig. 6.15 | Sorting an array's values into ascending order. (Part 3 of 3.)

```c
// Fig. 6.18: fig06_18.c
// Linear search of an array.
#include <stdio.h>
#define SIZE 100

// function prototype
size_t linearSearch(const int array[], int key, size_t size);

// function main begins program execution
int main(void)
{
   int a[SIZE]; // create array a

   // create some data
   for (size_t x = 0; x < SIZE; ++x) {
      a[x] = 2 * x;
   }

   printf("Enter integer search key: ");
   int searchKey; // value to locate in array a
   scanf("%d", &searchKey);

```

Fig. 6.18 | Linear search of an array. (Part 1 of 3.)

```
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
```

Fig. 6.18 | Linear search of an array. (Part 2 of 3.)

```
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  }
```

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

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

Fig. 6.18 | Linear search of an array. (Part 3 of 3.)

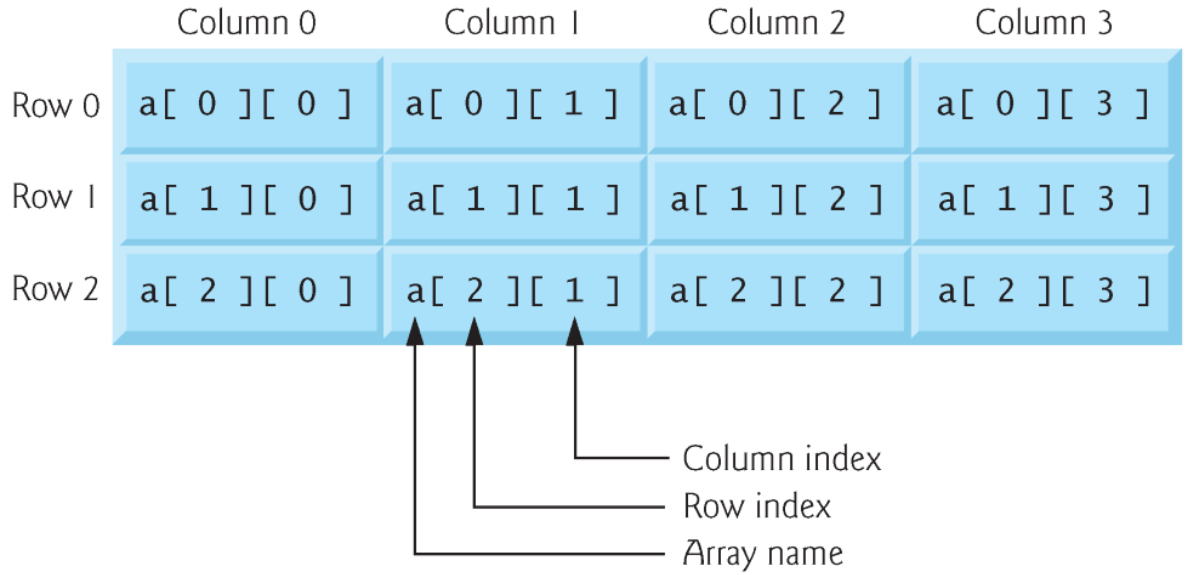

Fig. 6.20 | Two-dimensional array with three rows and four columns.

```c
// Fig. 6.21: fig06_21.c
// Initializing multidimensional arrays.
#include <stdio.h>

void printArray(int a[][3]); // function prototype

// function main begins program execution
int main(void)
{
   int array1[2][3] = {{1, 2, 3}, {4, 5, 6}};
   puts("Values in array1 by row are:");
   printArray(array1);

   int array2[2][3] = {1, 2, 3, 4, 5};
   puts("Values in array2 by row are:");
   printArray(array2);

   int array3[2][3] = {{1, 2}, {4}};
   puts("Values in array3 by row are:");
   printArray(array3);
}

```

Fig. 6.21 | Initializing multidimensional arrays. (Part 1 of 2.)

```
23  // function to output array with two rows and three columns
24  void printArray(int a[][3])
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 ", a[i][j]);
32        }
33
34        printf("\n"); // start new line of output
35     }
36  }
```

```
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
```

Fig. 6.21 | Initializing multidimensional arrays. (Part 2 of 2.)

```c
// Fig. 6.22: fig06_22.c
// Two-dimensional array manipulations.
#include <stdio.h>
#define STUDENTS 3
#define EXAMS 4

// function prototypes
int minimum(const int grades[][EXAMS], size_t pupils, size_t tests);
int maximum(const int grades[][EXAMS], size_t pupils, size_t tests);
double average(const int setOfGrades[], size_t tests);
void printArray(const int grades[][EXAMS], size_t pupils, size_t tests);

// function main begins program execution
int main(void)
{
   // initialize student grades for three students (rows)
   int studentGrades[STUDENTS][EXAMS] =
      { { 77, 68, 86, 73 },
        { 96, 87, 89, 78 },
        { 70, 90, 86, 81 } };

   // output array studentGrades
   puts("The array is:");
   printArray(studentGrades, STUDENTS, EXAMS);
```

Fig. 6.22 | Two-dimensional array manipulations. (Part 1 of 7.)

```
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, average(studentGrades[student], EXAMS));
35      }
36   }
37
```

Fig. 6.22 | Two-dimensional array manipulations. (Part 2 of 7.)

```
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
```

Fig. 6.22 | Two-dimensional array manipulations. (Part 3 of 7.)

```
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
```

Fig. 6.22 | Two-dimensional array manipulations. (Part 4 of 7.)

```
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
```

Fig. 6.22 | Two-dimensional array manipulations. (Part 5 of 7.)

```
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 }
```

Fig. 6.22 | Two-dimensional array manipulations. (Part 6 of 7.)

```
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
```

Fig. 6.22 | Two-dimensional array manipulations. (Part 7 of 7.)

```c
// Fig. 6.23: fig06_23.c
// Using variable-length arrays in C99
#include <stdio.h>

// function prototypes
void print1DArray(size_t size, int array[size]);
void print2DArray(int row, int col, int array[row][col]);

int main(void)
{
   printf("%s", "Enter size of a one-dimensional array: ");
   int arraySize; // size of 1-D array
   scanf("%d", &arraySize);

   int array[arraySize]; // declare 1-D variable-length array

   printf("%s", "Enter number of rows and columns in a 2-D array: ");
   int row1, col1; // number of rows and columns in a 2-D array
   scanf("%d %d", &row1, &col1);

   int array2D1[row1][col1]; // declare 2-D variable-length array

```

Fig. 6.23 | Using variable-length arrays in C99. (Part 1 of 5.)

```
23      printf("%s",
24         "Enter number of rows and columns in another 2-D array: ");
25      int row2, col2; // number of rows and columns in another 2-D array
26      scanf("%d %d", &row2, &col2);
27
28      int array2D2[row2][col2]; // declare 2-D variable-length array
29
30      // test sizeof operator on VLA
31      printf("\nsizeof(array) yields array size of %d bytes\n",
32         sizeof(array));
33
34      // assign elements of 1-D VLA
35      for (size_t i = 0; i < arraySize; ++i) {
36         array[i] = i * i;
37      }
38
39      // assign elements of first 2-D VLA
40      for (size_t i = 0; i < row1; ++i) {
41         for (size_t j = 0; j < col1; ++j) {
42            array2D1[i][j] = i + j;
43         }
44      }
45
```

Fig. 6.23 | Using variable-length arrays in C99. (Part 2 of 5.)

```
46      // assign elements of second 2-D VLA
47      for (size_t i = 0; i < row2; ++i) {
48         for (size_t j = 0; j < col2; ++j) {
49            array2D2[i][j] = i + j;
50         }
51      }
52
53      puts("\nOne-dimensional array:");
54      print1DArray(arraySize, array); // pass 1-D VLA to function
55
56      puts("\nFirst two-dimensional array:");
57      print2DArray(row1, col1, array2D1); // pass 2-D VLA to function
58
59      puts("\nSecond two-dimensional array:");
60      print2DArray(row2, col2, array2D2); // pass other 2-D VLA to function
61   }
62
63   void print1DArray(size_t size, int array[size])
64   {
65      // output contents of array
66      for (size_t i = 0; i < size; i++) {
67         printf("array[%d] = %d\n", i, array[i]);
68      }
69   }
```

Fig. 6.23 | Using variable-length arrays in C99. (Part 3 of 5.)

```
70
71  void print2DArray(size_t row, size_t col, int array[row][col])
72  {
73     // output contents of array
74     for (size_t i = 0; i < row; ++i) {
75        for (size_t j = 0; j < col; ++j) {
76           printf("%5d", array[i][j]);
77        }
78
79        puts("");
80     }
81  }
```

Fig. 6.23 | Using variable-length arrays in C99. (Part 4 of 5.)

```
Enter size of a one-dimensional array: 6
Enter number of rows and columns in a 2-D array: 2 5
Enter number of rows and columns in another 2-D array: 4 3

sizeof(array) yields array size of 24 bytes

One-dimensional array:
array[0] = 0
array[1] = 1
array[2] = 4
array[3] = 9
array[4] = 16
array[5] = 25

First two-dimensional array:
    0    1    2    3    4
    1    2    3    4    5

Second two-dimensional array:
    0    1    2
    1    2    3
    2    3    4
    3    4    5
```

Fig. 6.23 | Using variable-length arrays in C99. (Part 5 of 5.)

6.13 Secure C Programming

Bounds Checking for Array Indices

- It's important to ensure that every index you use to access an array element is within the array's bounds—that is, greater than or equal to 0 and less than the number of array elements.
- A two-dimensional array's row and column indices must be greater than or equal to 0 and less than the numbers of rows and columns, respectively.
- Allowing programs to read from or write to array elements outside the bounds of arrays are common security flaws.
- Reading from out-of-bounds array elements can cause a program to crash or even appear to execute correctly while using bad data.

6.13 Secure C Programming (Cont.)

- Writing to an out-of-bounds element (known as a *buffer overflow*) can corrupt a program's data in memory, crash a program and allow attackers to exploit the system and execute their own code.

- As we stated in the chapter, *C provides no automatic bounds checking for arrays*, so you must provide your own.

- For techniques that help you prevent such problems, see CERT guideline ARR30-C at www.securecoding.cert.org.

6.13 Secure C Programming (Cont.)

scanf_s

- Bounds checking is also important in string processing.
- When reading a string into a `char` array, `scanf` does not prevent buffer overflows.
- If the number of characters input is greater than or equal to the array's length, `scanf` will write characters—including the string's terminating null character (`'\0'`)—beyond the end of the array.
- This might *overwrite* other variables' values, and eventually the program might overwrite the string's `'\0'` if it writes to those other variables.

6.13 Secure C Programming (Cont.)

- Functions determine where strings end by looking for their terminating `'\0'` character.
- For example, function `printf` outputs a string by reading characters from the beginning of the string in memory and continuing until the string's `'\0'` is encountered.
- If the `'\0'` is missing, `printf` might read far beyond the end of the string until it encounters some other `'\0'` in memory.

6.13 Secure C Programming (Cont.)

- The C standard's optional Annex K provides new, more secure, versions of many string-processing and input/output functions, including `scanf_s`—a version of `scanf` that performs additional checks to ensure that it *does not* write beyond the end of a character array used to store a string.

- Assuming that `myString` is a 20-character array, the statement

```
scanf_s("%19s", myString, 20);
```

- reads a string into `myString`. Function `scanf_s` requires two arguments for each `%s` in the format string—a character array in which to place the input string and the number of array elements.

6.13 Secure C Programming (Cont.)

- The second of these arguments is used by `scanf_s` to prevent buffer overflows.
- For example, it's possible to supply a field width for `%s` that's too long for the underlying character array, or to simply omit the field width entirely.
- If the number of characters input plus the terminating null character is larger than the number of array elements, the `%s` conversion would fail.
- Because the preceding statement contains only one conversion specifier, `scanf_s` would return 0 indicating that no conversions were performed, and `myString` would be unaltered.
- In general, if your compiler supports the functions from the C standard's optional Annex K, you should use them.

6.13 Secure C Programming (Cont.)

Don't Use Strings Read from the User as Format-Control Strings

- You might have noticed that throughout this book, we never use single-argument printfs. Instead we use one of the following forms:
  - When we need to output a `'\n'` after the string, we use function puts (which automatically outputs a `'\n'` after its single string argument), as in

```
puts("Welcome to C!");
```

  - When we need the cursor to remain on the same line as the string, we use function `printf`, as in

```
printf("%s", "Enter first integer: ");
```

6.13 Secure C Programming (Cont.)

- Because we were displaying *string literals*, we certainly could have used the one-argument form of `printf`, as in

```
printf("Welcome to C!\n");
printf("Enter first integer: ");
```

- When `printf` evaluates the format-control string in its first (and possibly its only) argument, the function performs tasks based on the conversion specifier(s) in that string.

- If the format-control string were obtained from the user, an attacker could supply malicious conversion specifiers that would be “executed” by the formatted output function.

6.13 Secure C Programming (Cont.)

- It is possible to have issues with `printf()`, by using as format string a user-provided argument, i.e. `printf(arg)` instead of `printf("%s", arg)`.

- Since the caller did not push extra arguments, a string with some spurious `%` specifiers can be used to read whatever is on the stack, and with `%n` some values can be written to memory (`%n` means: "the next argument is an `int *`; go write there the number of characters emitted so far).

6.13 Secure C Programming (Cont.)

- Now that you know how to read strings into *character arrays*, it's important to note that you should never use as a `printf`'s format-control string a character array that might contain user input.

- For more information, see CERT guideline FIO30-C at www.securecoding.cert.org.