Data Structures: Arrays and Structs

Learning Objectives

- To learn how to declare and use arrays for storing collections of values of the same type
- To understand how to use a subscript to reference the individual values in an array
- To learn how to process the elements of an array in sequential order using loops
- To learn a method for searching an array
- To learn a method for sorting an array
- To become familiar with Big-O notation for estimating an algorithm's efficiency
- To learn how to declare and use structs to store collections of values of different types
- To understand how to use dot notation to reference the individual values in a struct
- To learn how to use multidimensional arrays for storing tables of data
- To understand how to use arrays whose elements are structs
- To learn how to use character arrays and C-style strings

SIMPLE DATA TYPES, whether built in (`int`, `float`, `bool`, `char`) or user defined (for example, enumeration type `day`), use a single variable to store a value. To solve many programming problems, it's more efficient to group data items together in main memory than to allocate a different variable to hold each item. For example, it's easier to write a program that processes exam scores for a class of 100 students if you can store all the scores in one area of memory and can access them as a group. C++ allows a programmer to group such

related data items together into a single composite structured variable. Without such a structured variable, we'd have to allocate 100 different variables to hold each individual score.

In this chapter, we look at two structured data types: the array and the struct. For an array, all items in the collection are the same data type, and we use a subscript to distinguish between them just as we use x_i to reference the ith item in a list. The struct, on the other hand, is an aggregate item that has components of different types. We also study arrays of arrays (multidimensional arrays) and arrays of structs.

9.1 The Array Data Type

array
A collection of data items stored under the same name.

array element
Each individual element in an array.

An **array** is a collection of data items associated with a particular name—for example, all the exam scores for a class of students could be associated with the name scores. We can reference each individual item—called an **array element**—in the array. We designate individual elements by using the array name and the element's position, starting with 0 for the first array element. The first element in the array named scores is referred to as `scores[0]`, the second element as `scores[1]`, and the tenth element as `scores[9]`. In general, the *k*th element in the array is referred to as `scores[k-1]`.

C++ stores an array in consecutive storage locations in main memory, one item per memory cell. We can perform some operations, such as passing the array as an argument to a function, on the whole array. We can also access individual array elements and process them like other simple variables.

Array Declaration

Arrays in C++ are specified using *array declarations* that specify the type, name, and size of the array:

```
float x[8];
```

C++ associates eight memory cells with the name x. Each element of array x may contain a single floating-point value. Therefore, a total of eight floating-point values may be stored and referenced using the array name x.

EXAMPLE 9.1

Assume `x` is the array shown in Figure 9.1. Table 9.1 shows some statements that manipulate the elements of this array. Figure 9.2 shows the array after the statements in Table 9.1 execute. Note that only `x[2]` and `x[3]` are changed.

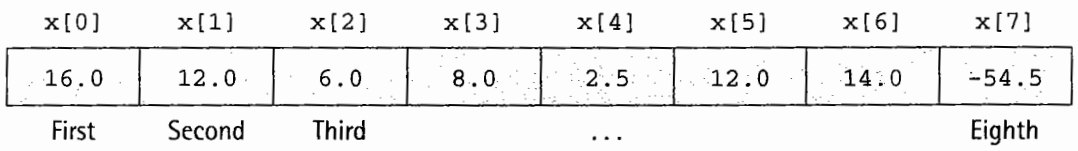


Figure 9.1 Array x

Table 9.1 Statements that Manipulate Elements of Array x in Figure 9.1

Statement	Explanation
`cout << x[0];`	Displays the value of `x[0]`, or 16.0.
`x[3] = 25.0;`	Stores the value 25.0 in `x[3]`.
`sum = x[0] + x[1];`	Stores the sum of `x[0]` and `x[1]`, or 28.0, in the variable sum.
`sum += x[2];`	Adds `x[2]` to sum. The new sum is 34.0.
`x[3] += 1.0;`	Adds 1.0 to `x[3]`. The new `x[3]` is 26.0.
`x[2] = x[0] + x[1];`	Stores the sum of `x[0]` and `x[1]` in `x[2]`. The new `x[2]` is 28.0.

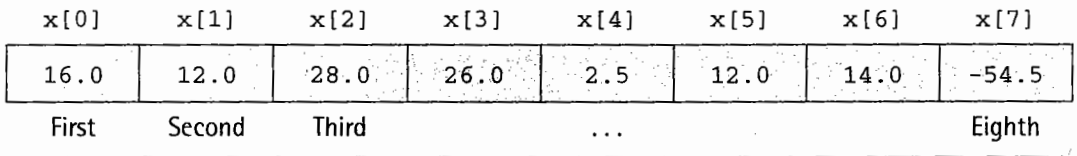


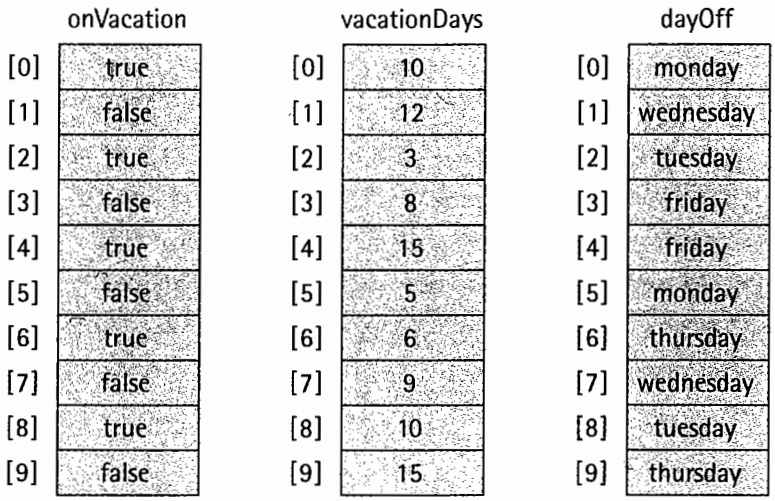
Figure 9.2 Array x after execution of statements in Table 9.1

EXAMPLE 9.2

The declarations for the plant operations program shown below allocate storage for four arrays, `onVacation`, `vacationDays`, `dayOff`, and `plantHours`.

```
const int NUM_EMP = 10;      // number of employees
bool onVacation[NUM_EMP];
int vacationDays[NUM_EMP];
enum day {sunday, monday, tuesday, wednesday, thursday,
          friday, saturday};
day dayOff[NUM_EMP];
float plantHours[7];
```

The arrays `onVacation` and `vacationDays` (Figure 9.3) have 10 elements, each with subscripts 0 through 9 (value of `NUM_EMP - 1`). Each element of array `onVacation` can store a type bool value. The contents of this array indicate which employees are on vacation (`onVacation[i]` is `true` if employee i is on vacation). If employees `0, 2, 4, 6,` and `8` are on vacation, the array will have the values shown in Figure 9.3. Array `vacationDays` shows the number of vacation days each employee has remaining.



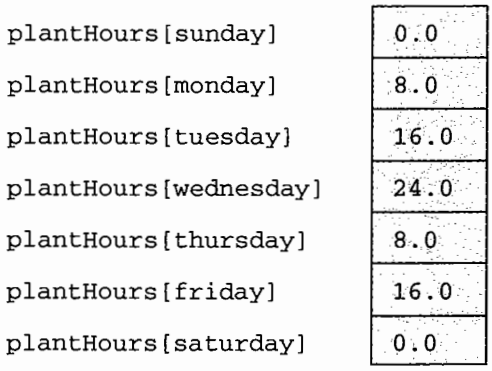
	onVacation	vacationDays	dayOff
[0]	true	10	monday
[1]	false	12	wednesday
[2]	true	3	tuesday
[3]	false	8	friday
[4]	true	15	friday
[5]	false	5	monday
[6]	true	6	thursday
[7]	false	9	wednesday
[8]	true	10	tuesday
[9]	false	15	thursday

Figure 9.3 Arrays `onVacation`, `vacationDays`, and `dayOff`

The array `dayOff` (Figure 9.3) also has 10 elements. Each element stores an enumerator from the enumeration type day, and the value of `dayOff[i]` indicates the weekday employee i has off. Because the data stored in `onVacation[i]`, `vacationDays[i]`, and `dayOff[i]` relate to the ith employee, the three arrays are called **parallel arrays.**

parallel arrays
Two or more arrays with the same number of elements used to store related information about a collection of objects.

The array `plantHours` (Figure 9.4) has seven elements with subscripts 0 through 6. We can use the subscripts 0 through 6 or the enumeration type day to reference these elements. The array element `plantHours[sunday]` (or `plantHours[0]`) indicates how many hours the plant was operating during Sunday of the past week. The array shown in Figure 9.4 indicates that the plant was closed on the weekend, operating single shifts on Monday and Thursday, double shifts on Tuesday and Friday, and a triple shift on Wednesday.



plantHours[sunday]	0.0
plantHours[monday]	8.0
plantHours[tuesday]	16.0
plantHours[wednesday]	24.0
plantHours[thursday]	8.0
plantHours[friday]	16.0
plantHours[saturday]	0.0

Figure 9.4 Array `plantHours`

Array Initialization

The next example shows how to specify the initial contents of an array as part of its declaration.

EXAMPLE 9.3

The statements below declare and initialize three arrays (see Figure 9.5). The list of initial values for each array is enclosed in braces and follows the assignment operator =. The first array element (subscript 0) stores the first value in each list, the second array element (subscript 1) stores the second value, and so on.

```
const int SIZE = 7;
int scores[SIZE] = {100, 73, 88, 84, 40, 97};
char grades[] = {'A', 'C', 'B', 'B', 'F', 'A'};
char myName[SIZE] = {'F', 'R', 'A', 'N', 'K'};
```

The length of the list of initial values can't exceed the size of the array. If the list contains fewer elements than the array size allows, then the value of any element not initialized is system dependent (indicated by ? in Figure 9.5). If the size of the array is not specified, indicated by an empty pair of brackets [ ], the compiler sets the array size to match the number of elements in the initial value list.

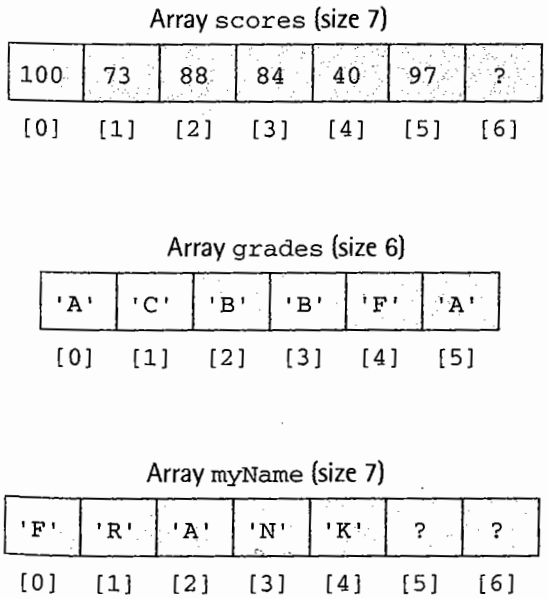


Figure 9.5 Initialization of three arrays

EXAMPLE 9.4

The array declaration

```
string kidNames[] = {"Richard", "Debbie", "Robin",
                     "Shelley", "Dara"};
```

allocates storage for an array of five strings called `kidNames`, storing the string literals shown in the initialization list in this array. The statement

```
cout << kidNames[0] << '/' << kidNames[4] << endl;
```

displays the output line

```
Richard/Dara
```

Array Declaration

Form: *element-type array-name*[*size*];
element-type array-name[*size*] = {*initialization-list*};

Example:
```
char myName[5];
float salaries[NUM_EMP];
char vowels[] = {'A', 'E', 'I', 'O', 'U'};
```

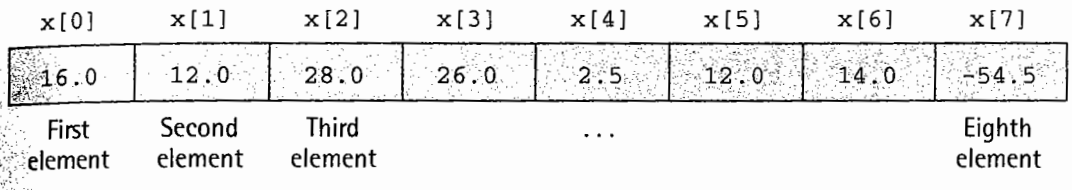
Interpretation: The identifier *array-name* describes a collection of array elements, each of which may be used to store data values of type *element-type*. The *size*, enclosed in brackets, [ ], specifies the number of elements contained in the array. The *size* value must be a constant expression consisting of constant values and constant identifiers. This value must be an integer and must be greater than or equal to 1. There is one array element for each value between 0 and the value *size* – 1.

In the initialized array declaration, the *size* in brackets is optional. In this case, the number of values in the *initialization-list* determines the array size. The *initialization-list* consists of constant expressions of type *element-type* separated by commas. Element 0 of the array being initialized is set to the first constant, element 1 to the second, and so on.

Array Subscripts

array subscript
A value or expression enclosed in brackets after the array name, specifying which element to access.
subscripted variable
A variable followed by a subscript in brackets, designating a particular array element.

To process the data stored in an array, we must be able to access its individual elements. We use the array name (a variable) and the **array subscript** to do this. The array subscript is enclosed in brackets after the array name and selects a particular array element for processing. For example, if x is the array with eight elements shown earlier in Figure 9.2 and repeated below, then the **subscripted variable** `x[0]` (read as "x sub 0") references the first element of the array `x`, `x[1]` the second element, and `x[7]` the eighth element. The number enclosed in brackets is the array subscript.



Understanding the difference between an array subscript value and an array element value is key. The subscripted variable `x[i]` references a particular element of array `x`. If i has the value `0`, the subscript value is `0`, and `x[0]` is referenced. The value of `x[i]` in this case is `16.0`. If i has the value `2`, the subscript value is `2`, and the value of `x[i]` is `28.0`.

A subscript can be an expression of any integral type (`int`, `long`, `short`, `char`, or an enumeration type). However, to create a valid reference, the value of this subscript must be between `0` and one less than the array size. If i has the value `8`, the subscript value is `8`, and we can't predict the value of `x[i]` because the subscript value is out of the allowable range.

EXAMPLE 9.5

Table 9.2 shows some simple statements involving the array `x` shown above and in Figure 9.2. In these statements, i is assumed to be type `int` with value `5`. Make sure you understand each statement. The two attempts to display element `x[10]`, which is not in the array, may result in a run-time error, but they're more likely to display incorrect results.

The last `cout` line in Table 9.2 uses `int(x[4])` as a subscript expression. Because this evaluates to `2`, the value of `x[2]` (*not* `x[4]`) is displayed. If the value of `int(x[4])` were outside the range `0` through `7`, its use as a subscript expression would reference a memory cell outside the array.

Two different subscripts are used in each of the three assignment statements at the bottom of the table. The first assignment statement copies the value of `x[6]` to `x[5]` (subscripts `i+1` and `i`); the second assignment statement copies the value of `x[5]` to `x[4]` (subscripts `i-1` and `i`). The last assignment statement causes a syntax error because there is an expression to the left of the assignment operator.

Array Subscript

Form: *name*[*subscript*]

Example: `x[3*i-2]`

Interpretation: The *subscript* must be an integral expression. Each time a subscripted variable is encountered in a program, the *subscript* is evaluated and its value determines which element of array *name* is referenced. The *subscript* value should be between the range 0 and one less than the array size (inclusive). If the *subscript* value is outside this range, a memory cell outside the array will be referenced.

Table 9.2 Some Simple Statements Referencing Array x in Figure 9.2

Statement	Effect
`cout << 3 << ' ' << x[3];`	Displays 3 and `26.0` (value of `x[3]`).
`cout << i << ' ' << x[i];`	Displays 5 and `12.0` (value of `x[5]`).
`cout << x[i] + 1;`	Displays `13.0` (value of `12.0 + 1`).
`cout << x[i] + i;`	Displays `17.0` (value of `12.0 + 5`).
`cout << x[i+1];`	Displays `14.0` (value of `x[6]`).
`cout << x[i+i];`	Value in `x[10]` is undefined.
`cout << x[2*i];`	Value in `x[10]` is undefined.
`cout << x[2*i-3];`	Displays `-54.5` (value of `x[7]`).
`cout << x[(int(x[4]))];`	Displays `28.0` (value of `x[2]`).
`x[i] = x[i+1];`	Assigns `14.0` (value of `x[6]`) to `x[5]`.
`x[i-1] = x[i];`	Assigns `14.0` (new value of `x[5]`) to `x[4]`.
`x[i] - 1 = x[i-1];`	Illegal assignment statement. Left side of an assignment operator must be a variable.

EXERCISES FOR SECTION 9.1

Self-Check

1. What is the difference between the expressions `x3` and `x[3]`?
2. For the following declarations, how many memory cells are reserved for data and what type of data can be stored there?
   a. `float prices[15];`
   b. `char grades[20];`
   c. `bool flags[10];`
   d. `int coins[5];`
   e. `string monthNames[12];`
3. Write array declarations for each of the following:
   a. Array `names` with element type `string`, size 100
   b. Array `checks` with element type `float`, size 20
   c. Array `madeBonus` with element type `bool`, size 7
   d. Array `daysInMonth` with element type `int`, size 12
4. Which of the following array declarations are legal? Defend your answer and describe the result of each legal declaration. For questions (a) and (e), use enumeration type `day` as defined in Example 9.2.
   a. `float payroll[friday];`
   b. `int workers[3] = {6, 8, 8, 0};`

c. `char vowels[] = {'a', 'e', 'i', 'o', 'u'};`

d. `int freq[12] = {0, 0, 3, 7, 10, 16, 28, 31};`

e. `day firstDay[] = {monday, friday, wednesday, saturday, friday};`

f. `string names [] = {"Harry", "Sally", "Don", "Sue"};`

Programming

1. Provide array type declarations for representing the following:

   a. A group of rooms (living room, dining room, kitchen, and so on) that have a given area

   b. A group of rooms as in part (a), but the array elements should indicate whether the room is carpeted

   c. Elementary school grade levels (0 through 6, where 0 means kindergarten) with a given number of students per grade

   d. A selection of colors (strings) representing the eye color of five people

   e. Answer (d) above assuming each array element stores an enumerator from the enumeration type below.

   ```
   typedef color = {blue, green, hazel, brown, black};
   ```

   f. An array `daysInMonth` with 31 (number of days in January) in element 0, 28 (number of days in February) in element 1, and so on.

9.2 Sequential Access to Array Elements

Many programs process all the elements of an array in sequential order, starting with the first element (subscript 0). To enter data into an array, print its contents, or perform other sequential processing tasks, use a for loop whose loop-control variable (`i`) is also the array subscript (`x[i]`). Increasing the value of the loop-control variable by 1 causes the next array element to be processed.

EXAMPLE 9.6

The array cube declared below stores the cubes of the first 10 integers (for example, `cube[1]` is 1, `cube[9]` is 729).

```
int cube[10];  // array of cubes
```

The `for` statement

```
for (int i = 0; i < 10; i++)
   cube[i] = i * i * i;
```

initializes this array as shown in Figure 9.6.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
0	1	8	27	64	125	216	343	512	729

Figure 9.6 Array cube

EXAMPLE 9.7

In Listing 9.1, the statements

```
const int MAX_ITEMS = 8;
float x[MAX_ITEMS];  // array of data
```

allocate storage for an array `x` with subscripts 0 through 7. The program uses three `for` loops to process the array. The loop-control variable `i`, with `i` in the range (`0 <= i <= 7`), is also the array subscript in each loop. The first `for` loop,

```
for (int i = 0; i < MAX_ITEMS; i++)
   cin >> x[i];
```

reads one data value into each array element (the first item is stored in `x[0]`, the second item in `x[1]`, and so on). The `cin` line executes for each value of `i` from 0 to 7; each repetition causes a new data value to be read and stored in `x[i]`. The subscript `i` determines the array element to receive the next data value. The data shown in the first line of the sample execution of Listing 9.1 cause the array to be initialized as shown in Figure 9.1.

Listing 9.1 Displaying a table of differences

```
// File: showDiff.cpp
// Computes the average value of an array of data and displays
//    the difference between each value and the average
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
   const int MAX_ITEMS = 8;
   float x[MAX_ITEMS];                  // array of data
   float average;                       // average value of data
   float sum;                           // sum of the data

   // Enter the data.
   cout << "Enter " << MAX_ITEMS << " numbers:";
   for (int i = 0; i < MAX_ITEMS; i++)
      cin >> x[i];
```

(continued)

Listing 9.1 Displaying a table of differences (continued)

```
    // Compute the average value.
    sum = 0.0;                              // initialize sum
    for (int i = 0; i < MAX_ITEMS; i++)
       sum += x[i];                         // add next element to sum
    average = sum / MAX_ITEMS;              // get average value

    cout << "The average value is " << average << endl << endl;

    // Display the difference between each item and the average.
    cout << "Table of differences between x[i] and the average."
         << endl;
    cout << setw(4) << "next" << setw(8) << "x[next]"
         << setw(14) << "difference" << endl;
    for (int i = 0; i < MAX_ITEMS; i++)
       cout << setw(4) << i << setw(8) << x[i]
            << setw(14) << (x[i] - average) << endl;

    return 0;
}
```

```
Enter 8 numbers: 16 12 6 8 2.5 12 14 -54.5
The average value is 2

Table of differences between x[i] and the average.
   i      x[i]    difference
   0        16            14
   1        12            10
   2         6             4
   3         8             6
   4       2.5           0.5
   5        12            10
   6        14            12
   7     -54.5         -56.5
```

The second `for` loop accumulates the sum of all eight elements of array `x` in the variable `sum`. Each time the `for` loop body repeats, the statement

```
sum += x[i];              // add next element to sum
```

adds the next element of array `x` to `sum`. Table 9.3 traces the first three loop iterations. The last `for` loop displays a table showing each array element, `x[i]`, and the difference between that element and the average value, `x[i] - average`.

Table 9.3 Trace of Second `for` Loop in Listing 9.1 (three iterations)

Statement Part	i	x[i]	sum	Effect
`sum = 0.0;`			0.0	Sets sum to zero
`for (i = 0; i < MAX_ITEMS; i++)`	0	16.0		Sets i to zero
`sum += x[i];`			16.0	Add x[0] to sum
increment and test i	1	12.0		1 < 8 is true
`sum += x[i];`			28.0	Add x[1] to sum
increment and test i	2	6.0		2 < 8 is true
`sum += x[i];`			34.0	Add x[2] to sum

Strings and Arrays of Characters

A `string` object uses an array whose elements are type `char` to store a character string. That's why the position of the first character of a string object is 0, not 1. If `name` is type `string`, we can use either `name[i]` or `name.at(i)` to access the `i`th character in string `name`.

EXAMPLE 9.8

The program in Listing 9.2 forms a *cryptogram*, or coded message, by replacing each letter in a message (a data string) with its code symbol. For example, in the simple next-letter-substitution code, we want to replace all `a`s with `b`, all `b`s with `c`, and so on. In Listing 9.2, the string constant `ALPHABET` contains all 26 lowercase letters and the string `CODE` contains the corresponding code symbols that appear under each letter.

The for loop heading

```
for (int i = 0; i < message.length(); i++)
```

processes each character in string message from the first to the last (subscript `message.length() - 1`). The statement

```
message[i] = CODE.at(pos);
```

replaces the letter in `message[i]` with its corresponding code symbol. This statement executes only if the value of `pos` is between `0` and `25`, or when function `find` is able to locate the lowercase form of `message[i]` in string `ALPHABET`. See Table 3.3 to review the string functions.

Listing 9.2 Cryptogram generator

```
// File: cryptogram.cpp
// Displays a cryptogram.

#include <string>
#include <iostream>
#include <cctype>
using namespace std;

int main()
{
   const string ALPHABET = "abcdefghijklmnopqrstuvwxyz";
   const string CODE =     "bcdefghijklmnopqrstuvwxyza";
   string message;         // message to encode
   char ch;                // next message character
   int pos;                // its position

   cout << "Enter a string to encode: ";
   getline(cin, message);

   // Encode message.
   for (int i = 0; i < message.length(); i++)
   {
      ch = tolower(message.at(i));     // ch to lowercase
      pos = ALPHABET.find(ch);         // find position of ch
      if ((pos >= 0) && (pos < 26))
         message[i] = CODE.at(pos);
   }

   cout << "The cryptogram is          : " << message << endl;

   return 0;
}
```

```
Enter a string to encode: This is a string.
The cryptogram is          : uijt jt b tusjoh.
```

EXERCISES FOR SECTION 9.2

Self-Check

1. **a.** If an array has 10 elements, what is displayed by the fragment below?

```
for (int i = 9; i >= 0; i--)
   cout << i << " " << x[i] * x[i] << ",";
   cout << endl;
```

b. What is displayed by the fragment below?

```
i = 0;
while (i < 10)
{
   cout << x[i] << " ";
   i += 2;
}
cout << endl;
```

2. The sequence of statements below changes the initial contents of array `x` displayed by the program in Listing 9.1. Describe what each statement does to the array and show the final contents of array `x` after all statements execute.

```
i = 2;
x[i] = x[i] + 10.0;
x[i-1] = x[2*i-1];
x[i+1] = x[2*i] + x[2*i+1];
for (int i = 4; i < 7; i++)
   x[i] = x[i+1];
for (int i = 2; i >= 0; i--)
   x[i+1] = x[i];
```

3. Trace the execution of the for loop in Listing 9.1 when the string read into `message` is `"4 Hearts."`

Programming

1. Write program statements that will do the following to array `x`, shown in Listing 9.1:

   a. Display all elements with odd subscripts on one line.

   b. Calculate the sum of the elements with odd subscripts only.

2. Write a loop that displays the characters in string `message` in reverse order.

3. Write a program to store in string `reverse` the characters in string `message` in reverse order and to display `"Is a palindrome"` if `message` and `reverse` contain the same string.

9.3 Array Arguments

The C++ operators (for example, <, ==, >, +, −) can be used to manipulate only one array element at a time. Consequently, an array name in an expression

will generally be followed by its subscript. The next example shows the use of array elements as function arguments.

Array Elements as Arguments

Example 9.9

You can use function `exchange` (see Listing 9.3) to switch the contents of its two floating-point arguments. The formal parameters `a1` and `a2` are used for both input and output and are therefore passed by reference.

This function may be used to exchange the contents of any two floating-point variables. The statement

```
exchange(x, y);
```

switches the original contents of the floating-point variables `x` and `y`.

This function may also be used to exchange the contents of any pair of elements of a floating-point array. For example, the function call

```
exchange(s[3], s[5]);
```

switches the contents of array elements `s[3]` and `s[5]` (see Figure 9.7). C++ treats the call to `exchange` in the same way as the earlier call involving `x` and `y`. In both cases, the contents of two floating-point memory locations (`s[3]` and `s[5]`) are exchanged.

Besides passing individual array elements to functions, we can write functions that have arrays as arguments. Such functions can manipulate some, or all, of the elements corresponding to an actual array argument.

Listing 9.3 Function to exchange the contents of two floating-point memory locations

```
void exchange
  (float& a1, float& a2)          // INOUT
{
   // Local data   . . .
   float temp;
   temp = a1;
   a1 = a2;
   a2 = temp;
}
```

s[0]	s[1]	s[2]	s[3]	s[4]	s[5]	s[6]	s[7]
16.0	12.0	28.0	26.0	2.5	12.0	14.0	-54.5

s[0]	s[1]	s[2]	s[3]	s[4]	s[5]	s[6]	s[7]
16.0	12.0	28.0	12.0	2.5	26.0	14.0	-54.5

Figure 9.7 Array `s` before (top) and after (bottom) exchange of `s[3]` and `s[5]`;

Passing an Array Argument

We can pass an entire array to a function by writing its name with no subscript in the argument list of a function call. What is actually stored in the function's corresponding formal parameter is the address of the initial array element. In the function body, we can use subscripts with the formal parameter to reference the array's elements. However, the function manipulates the actual array, not its own personal copy, so an assignment to one of the array elements by a statement in the function changes the contents of the actual array.

The next two examples illustrate the use of arrays as function arguments. We assume the calling function declares three floating-point arrays, `x`, `y`, and `z`, each of size `5`:

```
const int MAX_SIZE = 5;          // size of arrays
float x[MAX_SIZE];
float y[MAX_SIZE],
float z[MAX_SIZE];
```

EXAMPLE 9.10

Function **sameArray** in Listing 9.4 determines whether two arrays, represented by formal array parameters `a` and `b`, are identical. We consider two arrays to be identical if the first element of one is the same as the first element of the other, the second element of one is the same as the second element of the other, and so on.

We can determine that the arrays are not identical by finding a single pair of unequal elements. Before each iteration, the second part of the **while** loop condition

```
while ((i < size-1) && (a[i] == b[i]))
```

compares the `i`th elements of the actual arrays corresponding to formal array parameters `a` and `b`. Loop exit occurs when a pair of unequal elements is found or just before the last pair is tested.

Listing 9.4 Function sameArray

```
// File: sameArray.cpp
// Compares two float arrays for equality by comparing
//    corresponding elements

// Pre:  a[i] and b[i] (0 <= i <= size-1) are assigned
//       values.
// Post: Returns true if a[i] == b[i] for all i in range
//       0 through size - 1; otherwise, returns false.

bool sameArray
   (float a[],                  // IN: float arrays to be compared
    float b[],
    const int size)             // IN: size of the arrays
{
   // Local data  . . .
   int i;           // loop control variable and array subscript
   i = 0;
   while ((i < size-1) && (a[i] == b[i]))
      i++;

   return (a[i] == b[i]);       // define result
}
```

After loop exit, the statement

```
return (a[i] == b[i]);           // define result
```

defines the function result (true or false). If loop exit occurs because the pair of elements with subscript `i` is not equal, the function result is false. If loop exit occurs because the last pair of elements is reached (i equal to size-1), the function result will be true if these elements are equal and false if they're not.

As an example of how you might use function `sameArray`, the `if` statement

```
if (sameArray(x, y, MAX_SIZE))
   cout << "The arrays x and y are identical. " << endl;
else
   cout << "The arrays x and y are different. " << endl;
```

displays a message indicating whether the actual argument arrays `x` and `y` are identical.

Notice that we write the actual array `x` without brackets in the function call. Also, we declare formal array parameter `a` as `float a[]` in the parameter list for function `sameArray`. We discuss the reasons for this after the next example.

EXAMPLE 9.11

The assignment statement

```
z = x + y;      // invalid array operation
```

is invalid because the operator + can't be used with array arguments. However, we can use function `addArray` (see Listing 9.5) to add together, element by element, the contents of any two floating-point arrays of the same size. The function stores the sum of each pair of elements with subscript `i` in element `i` of a third floating-point array. The function call

```
addArray(MAX_SIZE, x, y, z);
```

causes the addition of corresponding elements in arrays `x` and `y` with the result stored in the corresponding element of array `z`.

Figure 9.8 shows the argument correspondence for arrays established by the function call

```
addArray(MAX_SIZE, x, y, z);
```

Because C++ always passes arrays by reference, arrays `a`, `b`, and `c` in the function `addArray` data area are represented by the addresses of the actual arrays `x`, `y`, and `z` used in the call. Thus the values of the elements of `x` and `y` used in the addition are taken directly from the arrays `x` and `y` in the calling function, and the function results are stored directly in array `z` in the calling function. After execution of the function, `z[0]` will contain the sum of `x[0]` and `y[0]`, or 3.8; `z[1]` will contain 6.7; and so on. Arrays `x` and `y` will be unchanged. In fact, the use of the reserved word `const` in front of the formal parameters `a` and `b` ensures that the contents of the corresponding actual arguments (`x` and `y`) can't be changed by the function. We use `const` in the declaration of formal parameter `size` for the same reason.

Notice that the formal parameter list in Listing 9.5 doesn't indicate how many elements are in the array parameters. Because C++ doesn't allocate space in memory for a copy of the actual array, the compiler doesn't need to know the size of an array parameter. In fact, since we don't provide the size, we have the flexibility to pass to function `addArray` three arrays of any size. In each call to `addArray`, the first argument tells the function how many elements to process. In the function prototype, we use `float[]` to represent each formal array parameter:

```
void addArray(int, const float[], const float[], float[]);
```

Listing 9.5 Function addArray

```
// File: addArray.cpp
// Stores the sum of a[i] and b[i] in c[i]

// Sums pairs of array elements with subscripts ranging from 0
       to size-1
// Pre:  a[i] and b[i] are defined (0 <= i <= size-1)
// Post: c[i] = a[i] + b[i] (0 <= i <= size-1)

void addArray
  (int size,                    // IN: the size of the arrays
   const float a[],             // IN: the first array
   const float b[],             // IN: the second array
   float c[])                   // OUT: result array
{
   // Add corresponding elements of a and b and store in c.
   for (int i = 0; i < size; i++)
      c[i] = a[i] + b[i];
}
```

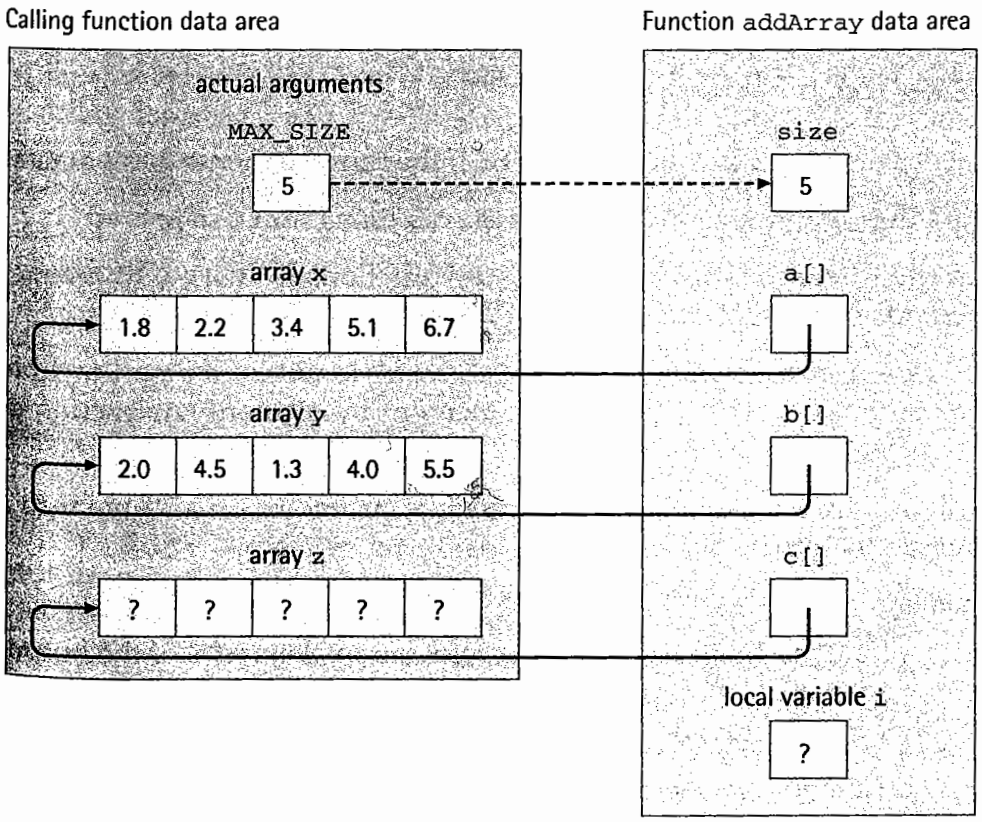


Figure 9.8 Argument correspondence for addArray(MAX_SIZE, x, y, z);

We summarize important points about arrays as function arguments next.

Arrays as Function Arguments

- In C++, arrays are passed by reference. Therefore, a formal array parameter in a function (`c` in Listing 9.5) represents the address of the first element of the actual array argument (`z` in Figure 9.8). All references to the formal array parameter are therefore references to the corresponding actual array argument in the calling function.
- In a function definition or a function prototype, we use empty brackets [ ] to inform the compiler that a formal parameter represents an array. We don't need to specify the size of this array because the compiler doesn't allocate storage for a copy of the actual array in the called function data area.
- The reserved word `const` indicates a formal array parameter that can't be changed by a function. If a function attempts to modify the contents of such an array, the compiler will generate an error message.
- In the function call, write the name of an actual argument array without using brackets after the array name.

EXERCISES FOR SECTION 9.3

Self-Check

1. In function `sameArray`, what will be the value of `i` when the statement

   ```
   return (a[i] == b[i]);
   ```

   executes if array `a` is equal to array `b`? If the third elements don't match?
2. Write the `return` statement for `sameArray` using an `if` statement.
3. Rewrite the `exchange` and `sameArray` functions to work with integer operands rather than floating-point operands.
4. Describe how to modify function `addArray` to obtain a new function, `subtractArray`, that performs an element-by-element subtraction of two integer arrays of the same size.
5. Explain why the size of an array must be passed as an argument to a function that processes an array.

Programming

1. Write a function that assigns a value of `true` to element `i` of the output array if element `i` of one input array has the same value as element `i` of the other input array; otherwise, assign a value of `false`.
2. Write a function `scalarMultArray` that multiplies an entire floating-point array `x` (consisting of n elements) by a single floating-point scalar `c`. Array `x` should be used for input/output.

9.4 Reading Part of an Array

Often a programmer doesn't know in advance exactly how many elements will be stored in an array. As an example, let's say you're writing a program to process exam scores. There may be 150 students in one section, 200 in the next, and so on. Because you must declare the array size before program execution begins, you must allocate enough storage space so that the program can process the largest expected array without error.

When you read the array data into memory, you should begin filling the array starting with the first element (at subscript 0) and be sure to keep track of how many data items are actually stored in the array. The part of the array that contains data is called the **filled subarray.** The *length* of the filled subarray is the number of data items that are actually stored in the array.

filled subarray
The portion of a partially filled array that contains data.

EXAMPLE 9.12

Function readScores in Listing 9.6 reads and stores in its array argument `scores` up to `MAX_SIZE` exam scores, where `MAX_SIZE` is an input argument that represents the size of the actual array corresponding to `scores`. The output argument `sectionSize` represents the length of the filled subarray and is initialized to zero. Within the loop, the statements

```
scores[sectionSize] = tempScore;   // save score just read
sectionSize++;
```

store the score just read (value of `tempScore`) in the next array element and increment `sectionSize`. After loop exit, the value of `sectionSize` is the length of the filled subarray `array`. The `if` statement displays a warning message when the array is filled (when `sectionSize` equals `MAX_SIZE`).

Listing 9.6 Function `readScores`

```
// File: readScores.cpp
// Reads an array of exam scores for a lecture section
//    of up to MAX_SIZE students.

// Pre: None
// Post: The data values are stored in array scores.
//       The number of values read is stored in sectionSize.
//       (0 <= sectionSize < MAX_SIZE).
```

(continued)

Listing 9.6 Function readScores (continued)

```
void readScores
  (int scores[],           // OUT: array to contain all scores read
   const int MAX_SIZE,     // IN: max size of array scores
   int& sectionSize)       // OUT: number of elements read
{
   // Local data  . . .
   const int SENTINEL = -1;        // sentinel value
   int tempScore;                  // temporary storage for each score

   // Read each array element until done.
   cout << "Enter next score after the prompt or enter "
        << SENTINEL << " to stop." << endl;
   sectionSize = 0;                // initial class size

   cout << "Score: ";
   cin >> tempScore;
   while ((tempScore != SENTINEL) && (sectionSize < MAX_SIZE))
   {
      scores[sectionSize] = tempScore;      // save score just read
      sectionSize++;
      cout << "Score: ";
      cin >> tempScore;
   }  // end while

   // Sentinel was read or array is filled.
   if (tempScore != SENTINEL)
   {
      cout << "Array is filled!" << endl;
      cout << tempScore << " not stored" << endl;
   }
}
```

Often when processing array data, we prefer to read the data from a file rather than type it in each time. Function `readScoresFile` (Listing 9.7) performs this operation assuming `ifstream` parameter `ins` is associated with a data file. We based this function on `readScores`. The major difference is the addition of parameter `ins` and the use of the `eof` function to detect the end of the data file instead of using a sentinel value. Also, we removed the prompts. If you prefer to use a sentinel value, the only change to function `readScores` in Listing 9.6 would be to add parameter `ins`, replace `cin` by `ins`, and delete all prompts.

Listing 9.7 Function readScoresFile

```
// File: readScoresFile.cpp
// Reads an array of exam scores for a lecture section
//    of up to MAX_SIZE students from a file.

// Pre: None
// Post: The data values are read from a file and stored
//       in array scores.
//       The number of values read is stored in sectionSize.
//       (0 <= sectionSize < MAX_SIZE).
void readScoresFile
  (ifstream& ins,        // IN: input stream of scores
   int scores[],         // OUT: array to contain all scores read
   const int MAX_SIZE,   // IN: max size of array scores
   int& sectionSize)     // OUT: number of elements read
{
   // Local data . . .
   int tempScore;              // temporary storage for each score

   // Read each array element until done.
   sectionSize = 0;            // initial class size
   ins >> tempScore;
   while (!ins.eof() && (sectionSize < MAX_SIZE))
   {
      scores[sectionSize] = tempScore;  // save score just read
      sectionSize++;
      ins >> tempScore;
   }  // end while

   // End of file reached or array is filled.
   if (!ins.eof())
   {
       cout << "Array is filled!" << endl;
       cout << tempScore << " not stored" << endl;
   }
}
```

EXERCISES FOR SECTION 9.4

Self-Check

1. Describe the changes necessary to use function readScores for reading floating-point data.

2. In function `readScores`, what prevents the user from entering more than `MAX_SIZE` scores?
3. Explain how you would modify function `readScoresFile` if the number of scores to read was included in the data file. Where should this number be placed?

Programming

1. Rewrite `readScoresFile` so that it stops reading when either a sentinel value is read, the end of the file is reached, or the array is filled.

9.5 Searching and Sorting Arrays

This section discusses two common problems in processing arrays: searching an array to determine the location of a particular value and sorting an array to rearrange the elements in an ordered fashion. As an example of an array search, we might want to search the array `scores` to determine which student, if any, got a particular score. An example of an array sort would be rearranging the array elements so that they're in increasing order by score. This would be helpful if we wanted to display the list in order by score or if we needed to locate several different scores in the array.

Finding the Smallest Value in an Array

We begin by solving a different kind of search problem: finding the smallest value in a subarray.

1. Assume that the first element is the smallest so far and save its subscript as "the subscript of the smallest element found so far."
2. For each array element after the first one
   2.1. If the current element < the smallest so far
      2.1.1. Save the subscript of the current element as "the subscript of the smallest element found so far."

Function `findIndexOfMin` in Listing 9.8 implements this algorithm for any subarray of floating-point elements. During each iteration of the loop, `minIndex` is the subscript of the smallest element so far and `x[minIndex]` is its value. The function returns the last value assigned to `minIndex`, which is the subscript of the smallest value in the subarray. Arguments `startIndex` and `endIndex` define the boundaries of the subarray, `x[startIndex]` through `x[endIndex]`, whose smallest value is being found. Passing these subscripts as arguments results in a more general function. To find the minimum element in the entire array, pass 0 to `startIndex` and the array size − 1 to `endIndex`.

Listing 9.8 Function `findIndexOfMin`

```
// File: arrayOperations.cpp
// Finds the subscript of the smallest value in a subarray.

// Returns the subscript of the smallest value in the subarray
// consisting of elements x[startindex] through x[endindex]
// Returns -1 if the subarray bounds are invalid.
// Pre: The subarray is defined and 0 <= startIndex <= endIndex.
// Post: x[minIndex] is the smallest value in the array.

int findIndexOfMin
    (const float x[],        // IN: array of elements
     int startIndex,         // IN: subscript of first element
     int endIndex)           // IN: subscript of last element
{
    // Local data . . .
    int minIndex;             // index of the smallest element found
    int i;                    // index of the current element

    // Validate subarray bounds
    if ((startIndex < 0) || (startIndex > endIndex))
    {
        cerr << "Error in subarray bounds" << endl;
        return -1;                          // return error indicator
    }

    // Assume the first element of subarray is smallest and check
    //   the rest.
    // minIndex will contain subscript of smallest examined so far.
    minIndex = startIndex;
    for (i = startIndex + 1; i <= endIndex; i++)
        if (x[i] < x[minIndex])
            minIndex = i;

    // Assertion: All elements are examined and minIndex is
    //    the index of the smallest element.
    return minIndex;                        // return result
} // end findIndexOfMin
```

Note that function `findIndexOfMin` returns the subscript (or index) of the smallest value, not the smallest value itself. Assuming `smallSub` is type `int`, and `yLength` is the number of array elements containing data, the following statements display the smallest value in array `y`.

```
smallSub = findIndexOfMin(y, 0, yLength - 1);
if (smallSub != -1)
   cout << "Value of smallest element in array y is "
        << y[smallSub] << endl;
else
   cerr << "Error - invalid array range" << endl;
```

PROGRAM STYLE Assertions as Comments

assertion
A comment that makes a statement about the program that must be true.

Normally we use a comment to document the purpose of the statements that follow it. In Listing 9.8, we introduced a different kind of comment called an **assertion**—a statement about the program that must be true at this point in the program. Instead of summarizing the intent of the statements that follow it, an assertion helps clarify the rationale for those statements. Programmers sometimes use assertions to help them prove that a program is correct. We'll use them to describe the expected status or state of the program.

Array Search

We can search an array for a particular element by comparing each array element, starting with the first (subscript 0), to the target, the value we're seeking. If a match occurs, we have found the target in the array and can return its subscript as a search result. If we test all array elements without finding a match, we return -1 to indicate that the target was not found. We choose -1 because no array element has a negative subscript.

INTERFACE FOR A SEARCH FUNCTION

Input Arguments

```
int items[]     // array to search
int size        // number of items to be examined
int target      // item to find
```

Output Arguments

(none)

Returns

Subscript of the first element of the array containing the target (return -1 if no element contains the target)

We use the linear search algorithm below and implemented in Listing 9.9 to search the input array for the target value.

1. For each array element
   1.1. If the current element contains the target
   1.2. Return the subscript of the current element.
2. Return -1.

Listing 9.9 The function `linSearch`

```
// File: arrayOperations.cpp
// Searches an integer array for a given element (the target)

// Array elements ranging from 0 to size -1 are searched for
//   an element equal to target.
// Pre:  The target and array are defined.
// Post: Returns the subscript of target if found;
//       otherwise, returns -1.

int linSearch
  (const int items[],        // IN: the array being searched
   int target,               // IN: the target being sought
   int size)                 // IN: the size of the array
{
   for (int next = 0; next < size; next++)
      if (items[next] == target)
         return next;             // found, return subscript

   // Assertion: All elements were tested without success.
   return -1;
}  // end linSearch
```

Sorting an Array in Ascending Order

Many programs execute more efficiently if the data they process are sorted before processing begins. For example, a check-processing program executes more quickly if all checks are in order by checking account number. Other programs produce more readable output if the information is sorted before it's displayed. For example, your university might want your instructor's grade report sorted by student ID number. In this section, we describe one simple sorting algorithm.

EXAMPLE 9.13

The *selection sort* is a fairly intuitive sorting algorithm. To perform a selection sort of an array of n elements (subscripts 0 through $n - 1$), we locate the smallest element in the array and then switch the smallest element with the element at subscript 0, thereby placing the smallest element at location 0. We then locate the smallest element remaining in the subarray with subscripts 1 through $n - 1$ and switch it with the element at subscript 1, thereby placing the second smallest element at location 1. We continue this process until all elements have been placed in their correct locations.

DATA REQUIREMENTS FOR A SORT FUNCTION

Input Arguments

```
float items[]  // array to sort
int n          // number of items to sort
```

Output Arguments

```
int items[]    // original array sorted in ascending order
```

Local Variables

```
int i          // subscript of first element in each subarray
int minSub     // subscript of each smallest item located by
               // findIndexOfMin
```

ALGORITHM

1. Starting with the first item in the array (subscript 0) and ending with the next-to-last item:
   - **1.1.** Set i equal to the subscript of the first item in the subarray to be processed in the next steps.
   - **1.2.** Find the subscript (`minSub`) of the smallest item in the subarray with subscripts ranging from i through `n-1`;
   - **1.3.** Exchange the smallest item found in step 1.2 with item i (exchange `items[minSub]` with `items[i]`).

Figure 9.9 traces the operation of the selection sort algorithm on an array with four elements. The first array shown is the original array. Then we show each step as the next smallest element is moved to its correct position. The shaded portion of each array represents the subarray that is sorted. We stop when the next to last element is placed in its correct position, so `n-1` exchanges will be required to sort an array with `n` elements.

We can use function `findIndexOfMin` (see again Listing 9.8) to perform step 1.2. Function `selSort` in Listing 9.10 implements the selection sort algorithm. Local variable `minSub` holds the index of the smallest value found so far in the current subarray. At the end of each pass, we call function `exchange` (see again Listing 9.3) to exchange the elements with subscripts `minSub` and `i`. After execution of function `selSort`, the array element values will be in increasing order.

The selection sort illustrates the importance of component reuse in problem solving and program engineering. We always strive to break down a complicated problem into simpler subproblems so that we can solve many of these subproblems using existing and tested components. Occasionally, we may need to make some minor modification to these components to adapt them to the current problem. But this is preferable to designing, implementing, and testing a new function from scratch.

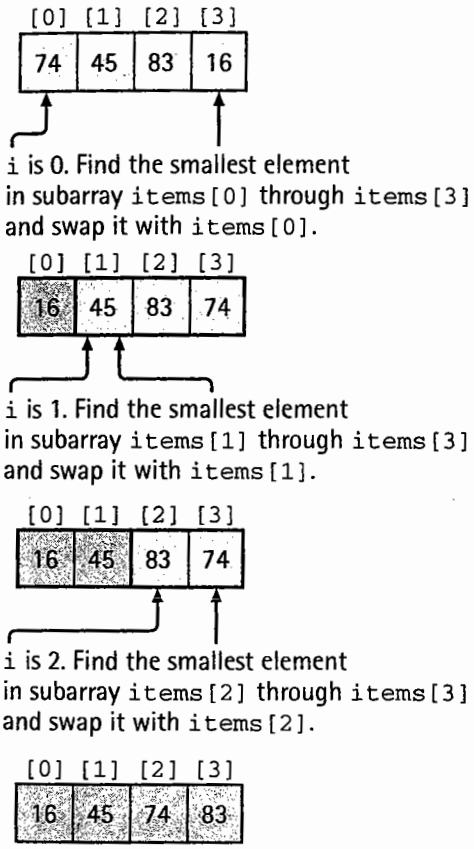


Figure 9.9 Trace of selection `sort`

Listing 9.10 Function `selSort`

```
// File: arrayOperations.cpp

// Sorts an array (ascending order) using selection sort algorithm
//    Uses exchange and findIndexOfMin
// Sorts the data in array items (items[0] through items[n-1]).
// Pre:  items is defined and n <= declared size of actual argument
//       array.
// Post: The values in items[0] through items[n-1] are in
//       increasing order.
void selSort(float items[], int n)
{
   // Local data . . .
   int minSub;  // subscript of each smallest item located by
                // findIndexOfMin

   for (int i = 0; i < n-1; i++)
   {
```

(continued)

Listing 9.10 Function `selSort` (continued)

```
        // Find index of smallest element in unsorted section of
        //    items.
        minSub = findIndexOfMin(items, i, n-1);

        // Exchange items at position minSub and i
        exchange(items[minSub], items[i]);
    }
}
```

EXERCISES FOR SECTION 9.5

Self-Check

1. For the linear search function in Listing 9.9, what happens if
   a. the last element of the array matches the target?
   b. several elements of the array match the target? The subscript of which one will be returned?
2. Trace the execution of the selection sort on the following list:

   `55 34 56 76 5 10 25 34`

   Show the array after each exchange occurs. How many comparisons are required? How many exchanges?
3. How could you modify the selection sort algorithm to arrange the data items in the array in descending order (smallest first)?
4. Modify the exchange function (Listing 9.3) to work for integer data.
5. In selection sort, there is no need to exchange the value in element i with the smallest value found if the smallest value is the same as the one at position i. How would you modify the algorithm to avoid this unnecessary exchange? How would this change affect the number of comparisons? How would it affect the number of exchanges?
6. Discuss how you could modify function `linSearch` to write a function `linSearchOrder` that performs a more efficient search of an array that has been sorted.

Programming

1. Write a function to count the number of items in an integer array having a value greater than `0`.
2. Another method of performing the selection sort is to place the largest value in position `n-1`, the next largest in `n-2`, and so on. Write this version.

3. Write function `linSearchLast` that finds the last occurrence of a target value in an array.
4. Rewrite function `selectSort` by writing step 1.2 of the algorithm (Find index of smallest item . . .) as a loop with loop control variable `j` instead of calling function `findIndexOfMin`.
5. Write function `linSearchOrder` described in Self-Check Exercise 6.

9.6 Analyzing Algorithms: Big-O Notation

Often we want to estimate the efficiency of an algorithm. In this section, we provide a brief introduction to the topic of algorithm analysis, focusing on the searching and sorting algorithms just introduced.

There are many algorithms for searching and sorting arrays. Because arrays can be very large, these operations can become time-consuming, so we need to know the relative efficiency of various algorithms for performing the same task. Because it's difficult to get a precise measure of the efficiency of an algorithm or program, we normally try to approximate the effect on an algorithm of a change in the number of items, n, that the algorithm processes. This way, we can see how an algorithm's execution time increases with n, so we can compare two algorithms by examining their growth rates.

Usually, growth rates are described in terms of the largest contributing factor as the value of n gets large (for example, as it grows to 1,000 or more). If we determine that the expression

$$2n^2 + n - 5$$

expresses the relationship between the processing time of an algorithm and n, we say that the algorithm is an $O(n^2)$ algorithm, where O is an abbreviation for "the order of magnitude." This notation is known as big-O notation. The reason that this is an $O(n^2)$ algorithm rather than an $O(2n^2)$ algorithm or an $O(2n^2 + n - 5)$ algorithm is that the dominant factor in the relationship is the n^2 term. For large values of n, the largest exponent term has by far the greatest impact on our measurements. For this reason, we tend to ignore the "smaller" terms and constants.

Analysis of a Search Algorithm

To search an array of n elements using the linear search function, we have to examine all n elements if the target is not present in the array. If the target is

in the array, then we have to search only until we find it. However, the target could be anywhere in the array—it's equally as likely to be at the beginning of the array as at the end. So, on average, we have to examine $n/2$ array elements to locate a target value in an array. This means that linear search is an $O(n)$ process; that is, the growth rate is linear with respect to the number of items being searched.

Analysis of a Sort Algorithm

To determine the efficiency of a sorting algorithm, we normally focus on the number of array element comparisons and exchanges that it requires. Performing a selection sort on an array of n elements requires $n - 1$ comparisons during the first pass, $n - 2$ during the second pass, and so on. Therefore, the total number of comparisons is represented by the series

$$1 + 2 + 3 + \ldots + (n - 2) + (n - 1)$$

The value of this series is

$$\frac{n \times (n - 1)}{2} = n^2/2 - n/2$$

At the end of each pass through the unsorted subarray, we exchange the smallest value with the array element at position `i`. Therefore, the number of exchanges is n and the growth rate for exchanges is $O(n)$.

Because the dominant term in the expression for the number of comparisons shown earlier is $n^2/2$, the selection sort is considered an $O(n^2)$ process and the growth rate is said to be quadratic (proportional to the square of the number of elements). What difference does it make whether an algorithm is an $O(n)$ or $O(n^2)$ process? Table 9.4 shows the evaluation of n and n^2 for different values of n. A doubling of n causes n^2 to increase by a factor of 4. Because n^2 increases much more rapidly than n, the performance of an $O(n)$ algorithm is not as adversely affected by an increase in array size as is an $O(n^2)$ algorithm. For large values of n (say, 100 or more), the difference in the performances of an $O(n)$ and an $O(n^2)$ algorithm is significant (see the last three lines of Table 9.4).

Other factors besides the number of comparisons and exchanges affect an algorithm's performance. For example, one algorithm may take more time preparing for each exchange or comparison than another. Also, one algorithm might exchange subscripts, whereas another might exchange the array elements themselves. The second process can be more time consuming. Another measure of efficiency is the amount of memory required by an algorithm. Further discussion of these issues is beyond the scope of this book.

Table 9.4 Table of Values of n and n^2

n	n^2
2	4
4	16
8	64
16	256
32	1024
64	4096
128	16384
256	65536
512	262144

EXERCISES FOR SECTION 9.6

Self-Check

1. Determine how many times the `cout` line is executed in each of the following fragments. Indicate whether the algorithm is $O(n)$ or $O(n^2)$.

   a.
   ```
   for (i = 0; i < n; i++)
      for (j = 0; j < n; j++)
         cout << i << ' ' << j;
   ```

   b.
   ```
   for (i = 0; i < n; i++)
      for (j = 1; j <= 2; j++)
         cout << i << ' ' << j;
   ```

   c.
   ```
   for (i = 0; i < n; i++)
      for (j = n; j > 0; j--)
         cout << i << ' ' << j;
   ```

   d.
   ```
   for (i = 0, i < n; i++)
      for (j = 0; j < i; j++)
         cout << i << " " << j << endl;
   ```

2. What does it mean for a term to dominate an equation? To get a feel for the answer, calculate the value of n, $n^2 + n - 5$, $n^2 - n + 5$, and $2n^2$ for values of $n = 1$, 10, 100, 1000, and 10000. You may want to write a small program to do this. Notice that each value of n is 10 times the previous value.

Programming

1. Write a program to compute and print the values of `y1` and `y2` (below) for n, from 10 to 1000 inclusive, in increments of 25. Does the result surprise you?

$$y1 = 100n + 10$$
$$y2 = 5n^2 + 2$$

9.7 Multidimensional Arrays

The array data structure allows a programmer to organize information in arrangements that are more complex than the linear or one-dimensional arrays discussed so far. We can declare and use arrays with several dimensions of different sizes.

Multidimensional Array Declaration

Syntax: *element-type aname*[$size_1$][$size_2$] . . . [$size_n$] ;

Example: `float table[NROWS][NCOLS];`

Interpretation: The array declaration allocates storage space for an array *aname* consisting of $size_1 \times size_2 \times \ldots \times size_n$ memory cells. Each memory cell can store one data item whose data type is specified by *element-type*. The individual array elements are referenced by the subscripted variables *aname*`[0][0] . . . [0]` through *aname*[$size_1 - 1$][$size_2 - 1$] . . . [$size_n - 1$]. An integer constant expression is used to specify each $size_i$.

Declaring Two-Dimensional Arrays

two-dimensional array
An array used to store information that is normally represented as a two-dimensional table.

Two-dimensional arrays, the most common multidimensional arrays, store information that we normally represent in table form. Let's first look at some examples of two-dimensional arrays—game boards, seating plans, and matrixes.

EXAMPLE 9.14

The array declaration

```
char ticTacToe[3][3];
```

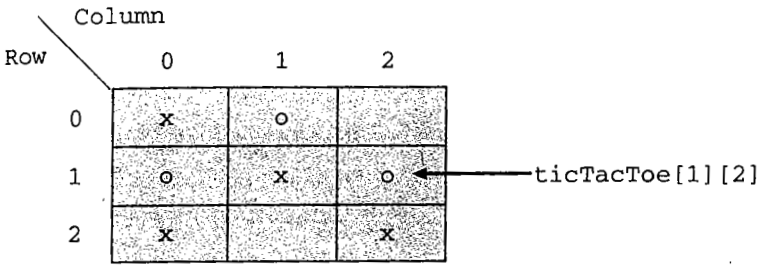


Figure 9.10 A `tic-tac-toe` board stored as array `ticTacToe`

allocates storage for a two-dimensional array (`ticTacToe`) with three rows and three columns. This array has nine elements, each of which must be referenced by specifying a row subscript (0, 1, or 2) and a column subscript (0, 1, or 2). Each array element contains a character value. For the array shown in Figure 9.10, the character 0 is stored in the subscripted variable

```
ticTacToe[1][2]
```

EXAMPLE 9.15

Your instructor wants to store the seating plan (Figure 9.11) for a classroom on a computer. The declarations

```
const int NUM_ROWS = 11;
const int SEATS_IN_ROW = 9;
string seatPlan[NUM_ROWS][SEATS_IN_ROW];
```

allocate storage for a two-dimensional array of strings called `seatPlan`. Array `seatPlan` could be used to store the first names of the students seated in a classroom with 11 rows and 9 seats in each row. The statement

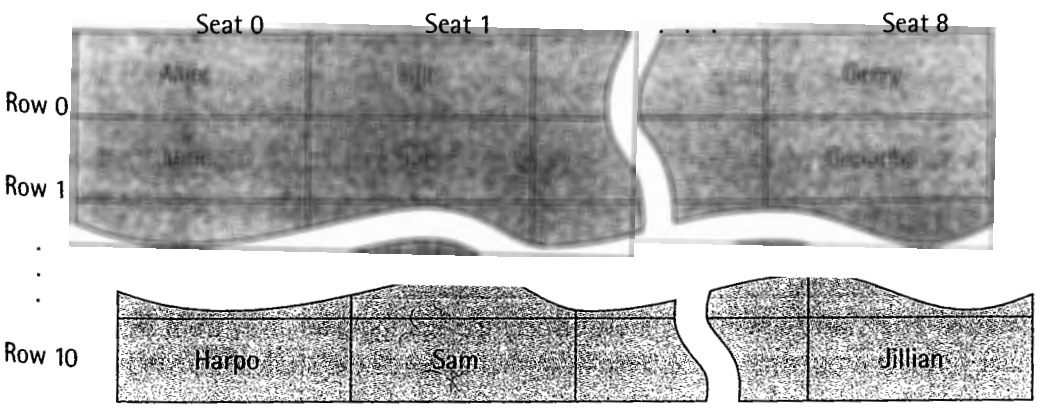


Figure 9.11 A classroom seating plan

```
seatPlan[0][8] = "Gerry";
```

stores the string "Gerry" in the last seat of the first row.

Initializing Two-Dimensional Arrays

EXAMPLE 9.16

The statements

```
const int NUM_ROWS = 2;
const int NUM_COLS = 3;
float matrix[NUM_ROWS][NUM_COLS] = {{5.0, 4.5, 3.0},
                                    {-16.0, -5.9, 0.0}};
```

allocate storage for the array matrix with two rows and three columns. In the initialization list enclosed in braces, each inner pair of braces contains the initial values for a row of the array matrix, starting with row 0, as shown below.

	col 0	col 1	col 2
row 0	5.0	4.5	3.0
row 1	-16.0	-5.9	0.0 ← matrix[1][2]

Nested Loops for Processing Two-Dimensional Arrays

You must use nested loops to access the elements of a two-dimensional array in row order or column order. If you want to access the array elements in row order (the normal situation), use the row subscript as the loop control variable for the outer loop and the column subscript as the loop control variable for the inner loop.

EXAMPLE 9.17

The nested `for` statements that follow display the seating plan array from Figure 9.11 in tabular form. The top row of the table lists the names of the students sitting in the last

row of the classroom; the bottom row of the table lists the names of the students sitting in the first row of the classroom (closest to the teacher).

```
for (int row = NUM_ROWS - 1; row >= 0; row--)
{
   for (int seat = 0; seat < SEATS_IN_ROW; seat++)
     cout << setw(10) << seatPlan[row][seat];
   cout << endl;
}
```

Two-Dimensional Arrays as Function Arguments

You can pass two-dimensional arrays as function arguments. In the function call, simply list the actual array name, just as you would for one-dimensional arrays. In the formal argument list, you must list the column dimension, but the row dimension is optional. For example, you could use the function prototype

```
float sumMatrix
  (float table[][NUM_COLS],     // IN: array to be summed
   int rows)                    // IN: number of rows
                                //       (rows > 0)
```

for function `sumMatrix` (see Listing 9.11), which calculates the sum of all the elements in an array of floating-point values with `NUM_COLS` (a constant) columns. The second function argument represents the number of rows to be included in the sum. You can use the statement

```
total = sumMatrix(matrix, NUM_COLS);
```

to assign to `total` the sum of the element values in the array `matrix` from Example 9.16.

You may be wondering why you only need to specify the column dimension (`NUM_COLS`) in the declaration of the formal array parameter `table`. Because the array elements are stored in row order, starting with row 0, C++ must know the number of elements in each row (value of `NUM_COLS`) in order to access a particular array element. For example, if `NUM_COLS` is 3, the first row is stored in array positions 0, 1, and 2, the

Listing 9.11 Function sumMatrix

```
// File: sumMatrix.cpp
// Calculates the sum of the elements in the first rows
//   of an array of floating point values
//   with NUM_COLS (a constant) columns.
// Pre: The type int constant NUM_COLS is defined (NUM_COLS > 0)
//      and the array element values are defined and rows > 0.
// Post: sum is the sum of all array element values.
// Returns: Sum of all values stored in the array.
float sumMatrix
  (float table[][NUM_COLS],   // IN: array to be summed
   int rows)                  // IN: number of rows in array
                              //        (rows > 0)
{
    float sum = 0.0;          // sum of all element values
                              //  - initially 0.0
    // Add each array element value to sum.
    for (int r = 0; r < rows; r++)
      for (int c = 0; c < NUM_COLS; c++)
        sum += table[r][c];

    return sum;
}
```

second row is stored in array positions 3, 4, and 5, and so on. C++ uses the formula

```
NUM_COLS * r + c
```

to compute the array position for array element `table[r][c]`. For example, array element `table[1][0]`, the first element in the second row, is stored at array position `3 * 1 + 0` (array position 3).

Arrays with Several Dimensions

C++ doesn't limit the number of dimensions an array may have, although arrays with more than three dimensions are rare. The array `sales` declared here

```
const int PEOPLE = 10;
const int YEARS = 5;
float sales[PEOPLE][YEARS][12];
```

is a three-dimensional array with 600 (value of 10 × 5 × 12) elements that may be used to store the monthly sales figures for the last five years for the 10 sales representatives in a company. Thus `sales[0][2][11]` represents the dollar amount of sales for the first salesperson (person subscript is 0) during December (month subscript is 11) of the third year (year subscript is 2). As you can see, three-dimensional and higher arrays consume memory space very quickly.

EXAMPLE 9.18

The following fragment finds and displays the total dollar amount of sales for each of the 10 sales representatives.

```
// Find and display the total dollar amount of sales by
//    person
for (int person = 0; person < PEOPLE; person++)
{
   totalSales = 0.0;
   // Find the total sales for the current person.
   for (int year = 0; year < YEARS; year++)
     for (int month = 0; month < 12; month++)
       totalSales += sales[person][year][month];

   cout << "Total sales amount for salesperson "
        << person << " is " << totalSales << endl;
}
```

Because we're displaying the total sales amount for each person, we use the salesperson subscript (`person`) as the loop control variable in the outermost loop. We set the value of `totalSales` to zero before executing the inner pair of nested loops. This pair accumulates the total sales amount for the current salesperson. The statement beginning with `cout` displays the accumulated total for each individual.

EXERCISES FOR SECTION 9.7

Self-Check

1. Assuming the following declarations

```
const int MAX_ROW = 9;
const int MAX_COL = 5;
float matrix[MAX_ROW][MAX_COL];
```

answer these questions:

a. How many elements are in the array `matrix`?

b. Write a statement to display the element in row 3, column 4.

c. How would you reference the element in the last column of the last row?

d. How would you reference the element in the exact middle of the array?

2. Write the declarations for a multidimensional array used to store the number of students enrolled in each section of the introductory programming course. Four different professors teach the course and each one teaches three sections. Answer this question if the course is taught at three campuses and there are five sections at each campus.

3. Explain why it isn't necessary to specify the first dimension (number of salespeople) when you declare a formal parameter that is an array with the same form as the one in Example 9.18. Describe the storage order for elements of this array in memory. Give a formula that will compute the position of array element `sales[i][j][k]`.

Programming

1. For the array `matrix` declared in Self-Check Exercise 1, do the following:

   a. Write a loop that computes the sum of the elements in row 4.

   b. Write a loop that computes the sum of the elements in column 3.

   c. Write nested loops that compute the sum of all the array elements.

   d. Write nested loops that display the array elements in the following order: display column 3 as the first output line, column 2 as the second output line, column 1 as the third output line, and column 0 as the fourth output line.

2. For the array `sales` in Example 9.18, do the following:

   a. Write a program fragment that displays the total sales amount in a table whose rows are years and whose columns are months.

   b. Write a program fragment that displays the total sales amount for each year and the total for all years.

9.8 The Struct Data Type

struct
A data structure that can be used to store a collection of related data items with different types.

Arrays are useful data structures for storing a collection of data elements of the same type. But many programming problems require us to store collections of related data that have different types. We can use a **struct** to accomplish this. For example, we can use a struct to store a variety of