مبانی برنامه نویسی
به زبان سی

۲۳، ۲۵ و ۲۷ آذر ۱۳۹۹

جلسه هفده، هجده و نوزده

ملکی مجد

مباحث این هفته:

- جستجو در یک آرایه
  - خطی
  - باینتری
- آرایه های چند بعدی
- اشاره گر
  - Passing Arguments to Functions
  - const Qualifier
  - Swap function
  - sizeof

6.8 Searching Arrays

- It may be necessary to determine whether an array contains a value that matches a certain key value.
  - The process of finding a particular element of an array is called searching.

- Here, we discuss two searching techniques—
  - the simple linear search technique and
  - the more efficient (but more complex) binary search technique.

6.8 Searching Arrays (Cont.)

- The linear search (Fig. 6.18) compares each element of the array with the search key.
  - Since the array is not in any particular order, it's just as likely that the value will be found in the first element as in the last.
  - *On average*, therefore, the program will have to compare the search key with *half the elements* of the array.

```c
   Linear search of an array */
#include <stdio.h>
#define SIZE 100

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

/* function main begins program execution */
int main( void )
{
   int a[ SIZE ]; /* create array a */
   int x; /* counter for initializing elements 0-99 of array a */
   int searchKey; /* value to locate in array a */
   int element; /* variable to hold location of searchKey or -1 */

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

   printf( "Enter integer search key:\n" );
   scanf( "%d", &searchKey );

   /* attempt to locate searchKey in array a */
   element = linearSearch( a, searchKey, SIZE );

   /* display results */
   if ( element != -1 ) {
      printf( "Found value in element %d\n", element );
   } /* end if */
   else {
      printf( "Value not found\n" );
   } /* end else */

   return 0; /* indicates successful termination */
} /* end main */
```

```
Enter integer search key:
36
Found value in element 18
```

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

```
39  /* compare key to every element of array until the location is found
40     or until the end of array is reached; return subscript of element
41     if key or -1 if key is not found */
42  int linearSearch( const int array[], int key, int size )
43  {
44     int n; /* counter */
45
46     /* loop through array */
47     for ( n = 0; n < size; ++n ) {
48
49        if ( array[ n ] == key ) {
50           return n; /* return location of key */
51        } /* end if */
52     } /* end for */
53
54     return -1; /* key not found */
55  } /* end function linearSearch */
```

6.8 Searching Arrays (Cont.)

- If the array is sorted, the high-speed binary search technique can be used.
  - The binary search algorithm eliminates from consideration one-half of the elements in a sorted array after each comparison.
  - The algorithm locates the middle element of the array and compares it to the search key.

6.8 Searching Arrays (Cont.)

- If they are **equal**, the search key is found and the array subscript of that element is returned.

- If they are **not equal**, the problem is reduced to searching one-half of the array.
  - If the search key is **less than the middle element** of the array, the **first** half of the array is searched, otherwise the **second** half of the array is searched.

  - If the search key is not found in the specified subarray (piece of the original array), the algorithm is repeated on one-quarter of the original array.

6.8 Searching Arrays (Cont.)

- The search continues until the search key is equal to the middle element of a subarray,
- or until the subarray consists of one element that is not equal to the search key (i.e., the search key is not found).

6.8 Searching Arrays (Cont.)

- In a worst case-scenario, searching an array of 1023 elements takes only 10 comparisons using a binary search.
  - Repeatedly dividing 1024 by 2 yields the values 512, 256, 128, 64, 32, 16, 8, 4, 2 and 1.
  - The number 1024 (2^{10}) is divided by 2 only 10 times to get the value 1.
  - Dividing by 2 is equivalent to one comparison in the binary search algorithm.
- An array of 1048576 (2^{20}) elements takes a maximum of 20 comparisons to find the search key. An array of one billion elements takes a maximum of 30 comparisons to find the search key.
- This is a tremendous increase in performance over the linear search that required comparing the search key to an average of half of the array elements.
  - For a one-billion-element array, this is a difference between an average of 500 million comparisons and a maximum of 30 comparisons!

Maximum comparisons for any array can be determined by finding the first power of 2 greater than the #array elements.

6.8 Searching Arrays (Cont.)

In following,

Iterative version of function `binarySearch`

- The function receives four arguments—an integer array `b` to be searched, an integer `searchKey`, the `low` array subscript and the `high` array subscript (these define the portion of the array to be searched).

- If the search key does not match the middle element of a subarray, the `low` subscript or `high` subscript is modified so that a smaller subarray can be searched.

6.8 Searching Arrays (Cont.)

- If the search key is less than the middle element, the `high` subscript is set to `middle - 1` and the search is continued on the elements from `low` to `middle - 1`.
- If the search key is greater than the middle element, the `low` subscript is set to `middle + 1` and the search is continued on the elements from `middle + 1` to `high`.

- The following program uses an array of 15 elements.
  - The first power of 2 greater than the number of elements in this array is 16 (2^4), so a maximum of 4 comparisons are required to find the search key.

```c
   Binary search of a sorted array */
#include <stdio.h>
#define SIZE 15

/* function prototypes */
int binarySearch( const int b[], int searchKey, int low, int high );
void printHeader( void );
void printRow( const int b[], int low, int mid, int high );
```

```c
/* function main begins program execution */
int main( void )
{
   int a[ SIZE ]; /* create array a */
   int i; /* counter for initializing elements 0-14
   int key; /* value to locate in array a */
   int result; /* variable to hold location of key

   /* create data */
   for ( i = 0; i < SIZE; i++ ) {
      a[ i ] = 2 * i;
   } /* end for */

   printf( "Enter a number between 0 and 28: " );
   scanf( "%d", &key );

   printHeader();

   /* search for key in array a */
   result = binarySearch( a, key, 0, SIZE - 1 );

   /* display results */
   if ( result != -1 ) {
      printf( "\n%d found in array element %d\n", key, result );
   } /* end if */
   else {
      printf( "\n%d not found\n", key );
   } /* end else */

   return 0; /* indicates successful termination */
} /* end main */
```

```
43  /* function to perform binary search of an array */
44  int binarySearch( const int b[], int searchKey, int low, int high )
45  {
46     int middle; /* variable to hold middle element of array */
47
48     /* loop until low subscript is greater than high subscript */
49     while ( low <= high ) {
50
51        /* determine middle element of subarray being searched */
52        middle = ( low + high ) / 2;
53
54        /* display subarray used in this loop iteration */
55        printRow( b, low, middle, high );
56
57        /* if searchKey matched middle element, return middle */
58        if ( searchKey == b[ middle ] ) {
59           return middle;
60        } /* end if */
61
62        /* if searchKey less than middle element, set new high */
63        else if ( searchKey < b[ middle ] ) {
64           high = middle - 1; /* search low end of array */
65        } /* end else if */
66
67        /* if searchKey greater than middle element, set new low */
68        else {
69           low = middle + 1; /* search high end of array */
70        } /* end else */
71     } /* end while */
72
73     return -1; /* searchKey not found */
74  } /* end function binarySearch */
```

```
76  /* Print a header for the output */
77  void printHeader( void )
78  {
79     int i; /* counter */
80
81     printf( "\nSubscripts:\n" );
82
83     /* output column head */
84     for ( i = 0; i < SIZE; i++ ) {
85        printf( "%3d ", i );
86     } /* end for */
87
88     printf( "\n" ); /* start new line of output */
89
90     /* output line of - characters */
91     for ( i = 1; i <= 4 * SIZE; i++ ) {
92        printf( "-" );
93     } /* end for */
94
95     printf( "\n" ); /* start new line of output */
96  } /* end function printHeader */
```

```
98  /* Print one row of output showing the current
99     part of the array being processed. */
100 void printRow( const int b[], int low, int mid, int high )
101 {
102    int i; /* counter for iterating through array b */
103
104    /* loop through entire array */
105    for ( i = 0; i < SIZE; i++ ) {
106
107       /* display spaces if outside current subarray range */
108       if ( i < low || i > high ) {
109          printf( "    " );
110       } /* end if */
111       else if ( i == mid ) { /* display middle element */
112          printf( "%3d*", b[ i ] ); /* mark middle value */
113       } /* end else if */
114       else { /* display other elements in subarray */
115          printf( "%3d ", b[ i ] );
116       } /* end else */
117    } /* end for */
118
119    printf( "\n" ); /* start new line of output */
120 } /* end function printRow */
```

```
Enter a number between 0 and 28: 25

Subscripts:
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
-------------------------------------------------------------
  0   2   4   6   8  10  12  14* 16  18  20  22  24  26  28
                                 16  18  20  22* 24  26  28
                                             24  26* 28
                                             24*

25 not found
```

```
Enter a number between 0 and 28: 8

Subscripts:
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
-------------------------------------------------------------
  0   2   4   6   8  10  12  14* 16  18  20  22  24  26  28
  0   2   4   6*  8  10  12
                  8  10* 12
                  8*

8 found in array element 4
```

```
Enter a number between 0 and 28: 6

Subscripts:
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
-------------------------------------------------------------
  0   2   4   6   8  10  12  14* 16  18  20  22  24  26  28
  0   2   4   6*  8  10  12

6 found in array element 3
```

6.9 Multiple-Subscripted Arrays

- Arrays in C can have **multiple subscripts**.
- A common use of multiple-subscripted arrays (also called multidimensional arrays) is to represent tables of values consisting of information arranged in rows and columns.
- To identify a particular table element, we must specify two subscripts: The first (by convention) identifies the element's row and the second (by convention) identifies the element's column.

6.9 Multiple-Subscripted Arrays (Cont.)

- Multiple-subscripted arrays can have more than two subscripts.

- In general, an array with *m rows and n columns is called an m-by-n array*

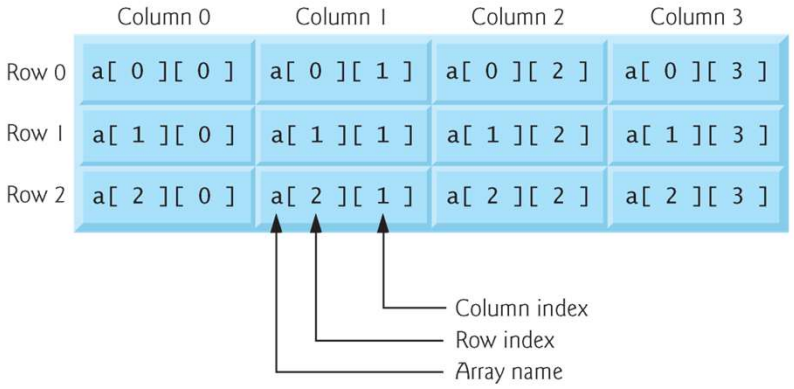
6.9 Multiple-Subscripted Arrays (Cont.)

- Consider a double-subscripted array, `a`.
  - The array contains three rows and four columns, so it's said to be a 3-by-4 array.

Every element in array `a` :

is identified by an element name of the form `a[i][j]`; `a` is the name of the array, and `i` and `j` are the subscripts that uniquely identify each element in `a`.

The names of the elements in the first row all have a first subscript of `0`; the names of the elements in the fourth column all have a second subscript of `3`.



6.9 Multiple-Subscripted Arrays (Cont.)

- A multiple-subscripted array can be initialized when it's defined, much like a single-subscripted array.
  - `int b[ 2 ][ 2 ] = { { 1, 2 }, { 3, 4 } };`
  - The values are grouped by row in braces.
  - The values in the first set of braces initialize row 0 and
  - the values in the second set of braces initialize row 1.
  - So, the values `1` and `2` initialize elements `b[0][0]` and `b[0][1]`, respectively, and the values `3` and `4` initialize elements `b[1][0]` and `b[1][1]`, respectively.

6.9 Multiple-Subscripted Arrays (Cont.)

- If there are not enough initializers for a given row, the remaining elements of that row are initialized to `0`.
- Thus,
  - `int b[ 2 ][ 2 ] = { { 1 }, { 3, 4 } };`

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

6.9 Multiple-Subscripted Arrays (Cont.)

- Figure 6.21 demonstrates defining and initializing double-subscripted arrays.
- The program defines three arrays of two rows and three columns (six elements each).
- The definition of `array1` (line 11) provides six initializers in two sublists.
- The first sublist initializes the first row (i.e., row 0) of the array to the values 1, 2 and 3; and the second sublist initializes the second row (i.e., row 1) of the array to the values 4, 5 and 6.

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

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

/* function main begins program execution */
int main( void )
{
   /* initialize array1, array2, array3 */
   int array1[ 2 ][ 3 ] = { { 1, 2, 3 }, { 4, 5, 6 } };
   int array2[ 2 ][ 3 ] = { 1, 2, 3, 4, 5 };
   int array3[ 2 ][ 3 ] = { { 1, 2 }, { 4 } };

   printf( "Values in array1 by row are:\n" );
   printArray( array1 );

   printf( "Values in array2 by row are:\n" );
   printArray( array2 );

   printf( "Values in array3 by row are:\n" );
   printArray( array3 );
   return 0; /* indicates successful termination */
} /* end main */
```

If the braces around each sublist are removed from the initializer list, the compiler initializes the elements of the first row followed by the elements of the second row.

Fig. 6.21 | Initializing multidimensional arrays. (Part 1 of 3.)

The first subscript of a multiple-subscripted array is not required either, but all subsequent subscripts are required.

```
25
26 /* function to output array with two rows and three columns */
27 void printArray( const int a[][ 3 ] )
28 {
29    int i; /* row counter */
30    int j; /* column counter */
31
32    /* loop through rows */
33    for ( i = 0; i <= 1; i++ ) {
34
35       /* output column values */
36       for ( j = 0; j <= 2; j++ ) {
37          printf( "%d ", a[ i ][ j ] );
38       } /* end inner for */
39
40       printf( "\n" ); /* start new line of output */
41    } /* end outer for */
42 } /* end function printArray */
```

```
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 3.)

6.9 Multiple-Subscripted Arrays (Cont.)

The first subscript of a multiple-subscripted array is not required either, but all subsequent subscripts are required.

- The compiler uses these subscripts to determine the locations in memory of elements in multiple-subscripted arrays.
- All array elements are stored consecutively in memory regardless of the number of subscripts.
  - In a double-subscripted array, the first row is stored in memory followed by the second row.

- Providing the subscript values in a parameter declaration enables the compiler to tell the function how to locate an element in the array.

6.9 Multiple-Subscripted Arrays (Cont.)

- In a double-subscripted array, each row is basically a single-subscripted array.
  - To locate an element in a particular row, the compiler **must know how many elements are in each row** so that it **can skip the proper number of memory locations** when accessing the array.
  - Thus, when accessing `a[1][2]` in our example, the compiler knows to skip the three elements of the first row to get to the second row (row 1).
  - Then, the compiler accesses the third element of that row (element 2).

6.9 Multiple-Subscripted Arrays (Cont.)

- Many common array manipulations use `for` repetition statements.
  - ```
    for ( column = 0; column <= 3; column++ ) {
       a[ 2 ][ column ] = 0;
    }
    ```
- is equivalent to the assignment statements:
  - ```
    a[ 2 ][ 0 ] = 0;
    a[ 2 ][ 1 ] = 0;
    a[ 2 ][ 2 ] = 0;
    a[ 2 ][ 3 ] = 0;
    ```

6.9 Multiple-Subscripted Arrays (Cont.)

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

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

6.9 Multiple-Subscripted Arrays (Cont.)

- Sample problem
  - several array manipulations on 3-by-4 array `studentGrades`
  - Each row of the array represents a student and each column represents a grade on one of the four exams the students took during the semester.

```c
   Double-subscripted array example */
#include <stdio.h>
#define STUDENTS 3
#define EXAMS 4

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

/* function main begins program execution */
int main( void )
{
   int student; /* student counter */

   /* initialize student grades for three students (rows) */
   const int studentGrades[ STUDENTS ][ EXAMS ] =
      { { 77, 68, 86, 73 },
        { 96, 87, 89, 78 },
        { 70, 90, 86, 81 } };
   /* output array studentGrades */
   printf( "The array is:\n" );
   printArray( studentGrades, STUDENTS, EXAMS );

   /* determine smallest and largest grade values */
   printf( "\n\nLowest grade: %d\nHighest grade: %d\n",
      minimum( studentGrades, STUDENTS, EXAMS ),
      maximum( studentGrades, STUDENTS, EXAMS ) );

   /* calculate average grade for each student */
   for ( student = 0; student < STUDENTS; student++ ) {
      printf( "The average grade for student %d is %.2f\n",
         student, average( studentGrades[ student ], EXAMS ) );
   } /* end for */

   return 0; /* indicates successful termination */
} /* end main */
```

a double-subscripted array is basically an array of single-subscripted arrays and that the name of a single-subscripted array is the address of the array in memory

determines the lowest grade of any student for the semester.

```
42  /* Find the minimum grade */
43  int minimum( const int grades[][ EXAMS ], int pupils, int tests )
44  {
45     int i; /* student counter */
46     int j; /* exam counter */
47     int lowGrade = 100; /* initialize to highest possible grade */
48
49     /* loop through rows of grades */
50     for ( i = 0; i < pupils; i++ ) {
51
52        /* loop through columns of grades */
53        for ( j = 0; j < tests; j++ ) {
54
55           if ( grades[ i ][ j ] < lowGrade ) {
56              lowGrade = grades[ i ][ j ];
57           } /* end if */
58        } /* end inner for */
59     } /* end outer for */
60
61     return lowGrade; /* return minimum grade */
62  } /* end function minimum */
63
```

determines the highest grade of any student for the semester

```
64  /* Find the maximum grade */
65  int maximum( const int grades[][ EXAMS ], int pupils, int tests )
66  {
67     int i; /* student counter */
68     int j; /* exam counter */
69     int highGrade = 0; /* initialize to lowest possible grade */
70
71     /* loop through rows of grades */
72     for ( i = 0; i < pupils; i++ ) {
73
74        /* loop through columns of grades */
75        for ( j = 0; j < tests; j++ ) {
76
77           if ( grades[ i ][ j ] > highGrade ) {
78              highGrade = grades[ i ][ j ];
79           } /* end if */
80        } /* end inner for */
81     } /* end outer for */
82
83     return highGrade; /* return maximum grade */
84  } /* end function maximum */
85
```

determines a particular student's semester average

```
86  /* Determine the average grade for a particular student */
87  double average( const int setOfGrades[], int tests )
88  {
89     int i; /* exam counter */
90     int total = 0; /* sum of test grades */
91
92     /* total all grades for one student */
93     for ( i = 0; i < tests; i++ ) {
94        total += setOfGrades[ i ];
95     } /* end for */
96
97     return ( double ) total / tests; /* average */
98  } /* end function average */
99
```

outputs the double-subscripted array in a neat, tabular format

```
100 /* Print the array */
101 void printArray( const int grades[][ EXAMS ], int pupils, int tests )
102 {
103    int i; /* student counter */
104    int j; /* exam counter */
105
106    /* output column heads */
107    printf( "                [0]  [1]  [2]  [3]" );
108
109    /* output grades in tabular format */
110    for ( i = 0; i < pupils; i++ ) {
111
112       /* output label for row */
113       printf( "\nstudentGrades[%d] ", i );
114
115       /* output grades for one student */
116       for ( j = 0; j < tests; j++ ) {
117          printf( "%-5d", grades[ i ][ j ] );
118       } /* end inner for */
119    } /* end outer for */
120 } /* end function printArray */
```

```
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
```

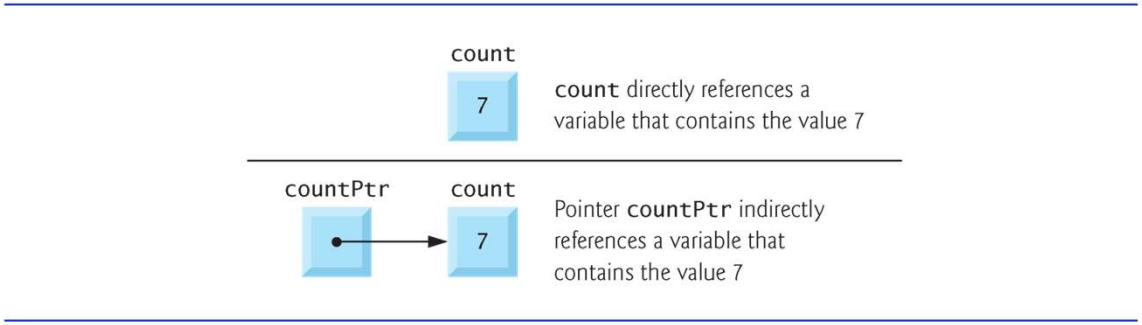
7.2 Pointer Variable Definitions and Initialization

- Pointers are variables whose values are memory addresses.

- Normally, a variable directly contains a specific value.
- A pointer, on the other hand, contains an address of a variable that contains a specific value.

7.2 Pointer Variable Definitions and Initialization

- A variable name directly references a value, and a pointer indirectly references a value.
  - Referencing a value through a pointer is called indirection.



7.2 Pointer Variable Definitions and Initialization (cont.)

- Pointers, like all variables, must be defined before they can be used.
- The definition
  - `int *countPtr, count;`

  specifies that variable `countPtr` is of type `int *` (i.e., a pointer to an integer) and is read, "`countPtr` is a pointer to `int`" or "`countPtr` points to an object of type `int`." Also, the variable `count` is defined to be an `int`, not a pointer to an `int`.
- The * only applies to `countPtr` in the definition.
- When * is used in this manner in a definition, it indicates that the variable being defined is a pointer.
- Pointers can be defined to point to objects of any type.

Common Programming Error 7.1

*The asterisk (*) notation used to declare pointer variables does not distribute to all variable names in a declaration. Each pointer must be declared with the * prefixed to the name; e.g., if you wish to declare* `xPtr` *and* `yPtr` *as* `int` *pointers, use* `int *xPtr, *yPtr;`.

7.2 Pointer Variable Definitions and Initialization (cont.)

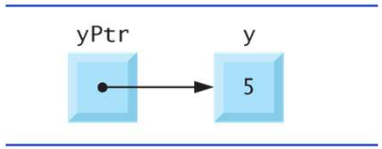
- Pointers should be initialized either when they're defined or in an assignment statement.
- A pointer may be initialized to `NULL`, `0` or an address.
- A pointer with the value `NULL` points to nothing.
- `NULL` is a symbolic constant defined in the `<stddef.h>` header (and several other headers, such as `<stdio.h>`).
- Initializing a pointer to `0` is equivalent to initializing a pointer to `NULL`, but `NULL` is preferred.
  - When `0` is assigned, it's first converted to a pointer of the appropriate type.
    - The value `0` is the only integer value that can be assigned directly to a pointer variable.

7.3 Pointer Operators

- The &, or address operator, is a unary operator that returns the address of its operand.
- For example, assuming the definitions
  - ```
    int y = 5;
    int *yPtr;
    ```

  the statement
  - ```
    yPtr = &y;
    ```

  assigns the address of the variable `y` to pointer variable `yPtr`.



- Variable `yPtr` is then said to "point to" `y`.

7.3 Pointer Operators (Cont.)

- Below, shows the representation of the pointer in memory, assuming that integer variable `y` is stored at location 600000, and pointer variable `yPtr` is stored at location 500000.

- The operand of the address operator must be a variable; the address operator cannot be applied to constants, to expressions or to variables declared with the storage-class `register`.

7.3 Pointer Operators (Cont.)

- The unary * operator,

  commonly referred to as the indirection operator or dereferencing operator, returns the value of the object to which its operand (i.e., a pointer) points.

  For example, the statement

  - `printf( "%d", *yPtr );`

  prints the value of variable `y`, namely 5.

- Using * in this manner is called dereferencing a pointer.

Common Programming Error 7.3

Dereferencing a pointer that has not been properly initialized or that has not been assigned to point to a specific location in memory is an error. This could cause a fatal execution-time error, or it could accidentally modify important data and allow the program to run to completion with incorrect results.

```c
/* Fig. 7.4: fig07_04.c
   Using the & and * operators */
#include <stdio.h>

int main( void )
{
   int a; /* a is an integer */
   int *aPtr; /* aPtr is a pointer to an integer */

   a = 7;
   aPtr = &a; /* aPtr set to address of a */

   printf( "The address of a is %p"
           "\nThe value of aPtr is %p", &a, aPtr );

   printf( "\n\nThe value of a is %d"
           "\nThe value of *aPtr is %d", a, *aPtr );

   printf( "\n\nShowing that * and & are complements of "
           "each other\n&*aPtr = %p"
           "\n*&aPtr = %p\n", &*aPtr, *&aPtr );
   return 0; /* indicates successful termination */
} /* end main */
```

Fig. 7.4 | Using the & and * pointer operators. (Part 1 of 2.)

& and * operators are complements of one another when they're both applied consecutively to `aPtr` in either order (line 21), the same result is printed

```
The address of a is 0012FF7C
The value of aPtr is 0012FF7C

The value of a is 7
The value of *aPtr is 7

Showing that * and & are complements of each other.
&*aPtr = 0012FF7C
*&aPtr = 0012FF7C
```

precedence and associativity of the operators introduced to this point

Operators	Associativity	Type
`()` `[]`	left to right	highest
`+` `-` `++` `--` `!` `*` `&` (*type*)	right to left	unary
`*` `/` `%`	left to right	multiplicative
`+` `-`	left to right	additive
`<` `<=` `>` `s>=`	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. 7.5 | Operator precedence and associativity.

7.4 Passing Arguments to Functions by Reference

- There are two ways to pass arguments to a function
  - call-by-value
  - call-by-reference.
- All arguments in C are passed by value.
- Many functions require the capability to modify one or more variables in the caller or to pass a pointer to a large data object to avoid the overhead of passing the object by value (which incurs the overhead of making a copy of the object).

For these purposes, C provides the capabilities for simulating call-by-reference.

By using pointers and the indirection operator

7.4 Passing Arguments to Functions by Reference (Cont.)

- When calling a function with arguments that should be modified, **the addresses of the arguments are passed**.

  This is normally accomplished by applying the address operator (&) to the variable (in the caller) whose value will be modified.

- As we saw before, arrays are not passed using operator & because C automatically passes the starting location in memory of the array

  (the name of an array is equivalent to `&arrayName[0]`).

- When the address of a variable is passed to a function, the indirection operator (*) may be used in the function to modify the value at that location in the caller's memory.

7.4 Passing Arguments to Functions by Reference (Cont.)

- In what follows,
- There are two versions of a function that cubes an integer
  - `cubeByValue` and
  - `cubeByReference`.

```c
/* Fig. 7.6: fig07_06.c
   Cube a variable using call-by-value */
#include <stdio.h>

int cubeByValue( int n ); /* prototype */

int main( void )
{
   int number = 5; /* initialize number */

   printf( "The original value of number is %d", number );

   /* pass number by value to cubeByValue */
   number = cubeByValue( number );

   printf( "\nThe new value of number is %d\n", number );
   return 0; /* indicates successful termination */
} /* end main */

/* calculate and return cube of integer argument */
int cubeByValue( int n )
{
   return n * n * n; /* cube local variable n and return result */
} /* end function cubeByValue */
```

number

n

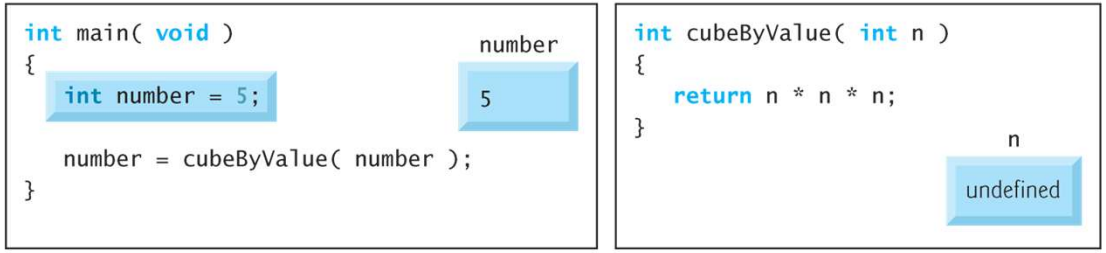
```
The original value of number is 5
The new value of number is 125
```

```c
/* Fig. 7.7: fig07_07.c
   Cube a variable using call-by-reference with a pointer argument */

#include <stdio.h>

void cubeByReference( int *nPtr ); /* prototype */

int main( void )
{
   int number = 5; /* initialize number */

   printf( "The original value of number is %d", number );

   /* pass address of number to cubeByReference */
   cubeByReference( &number );

   printf( "\nThe new value of number is %d\n", number );
   return 0; /* indicates successful termination */
} /* end main */

/* calculate cube of *nPtr; modifies variable number in main */
void cubeByReference( int *nPtr )
{
   *nPtr = *nPtr * *nPtr * *nPtr; /* cube *nPtr */
} /* end function cubeByReference */
```

header specifies that cubeByReference receives the address of an integer variable as an argument


```
The original value of number is 5
The new value of number is 125
```

Step 1: Before `main` calls `cubeByValue`:



Step 2: After `cubeByValue` receives the call:

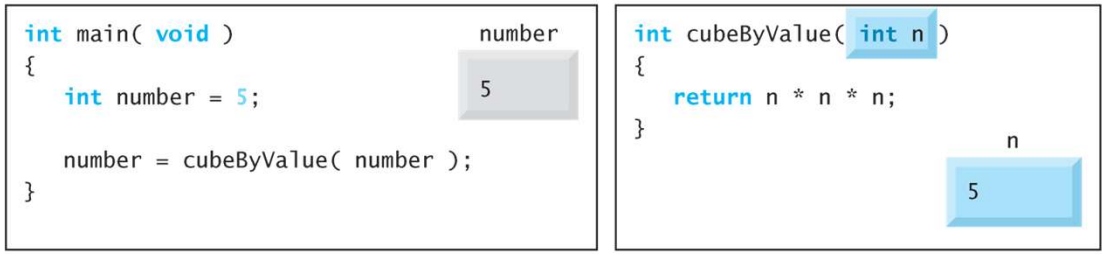
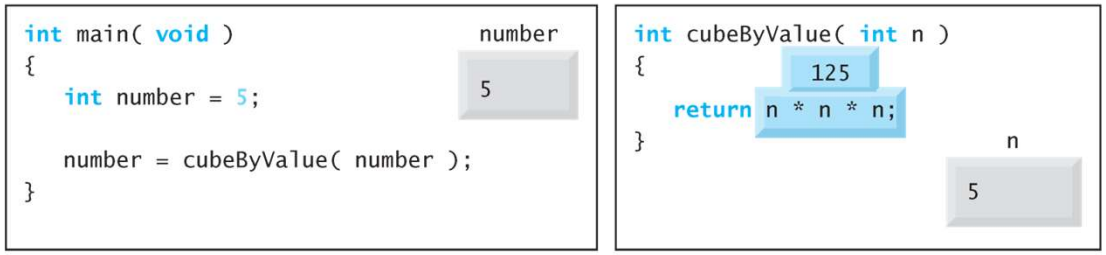


Fig. 7.8 | Analysis of a typical call-by-value. (Part 1 of 3.)

Step 3: After `cubeByValue` cubes parameter `n` and before `cubeByValue` returns to `main`:



Step 4: After `cubeByValue` returns to `main` and before assigning the result to `number`:

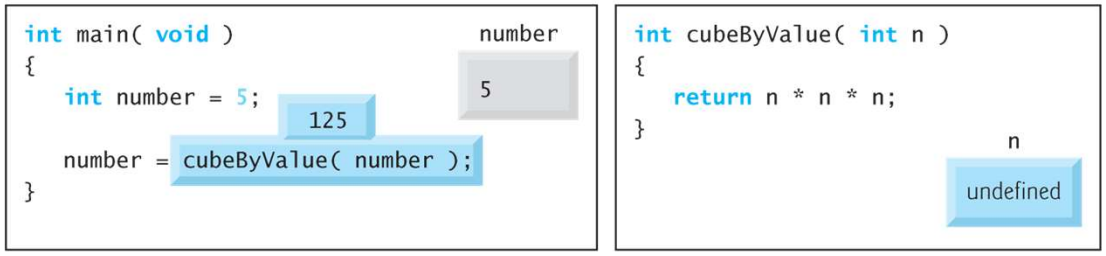


Fig. 7.8 | Analysis of a typical call-by-value. (Part 2 of 3.)

Step 5: After `main` completes the assignment to `number`:

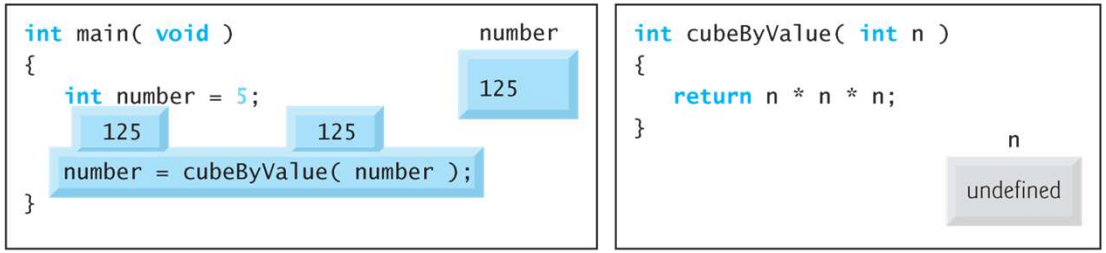
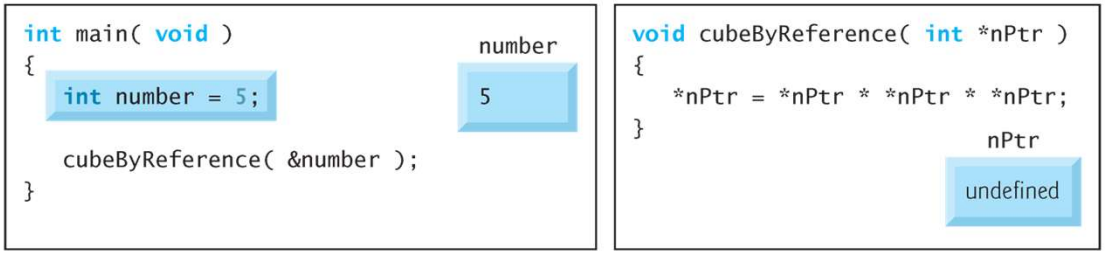


Fig. 7.8 | Analysis of a typical call-by-value. (Part 3 of 3.)

Step 1: Before `main` calls `cubeByReference`:



Step 2: After `cubeByReference` receives the call and before `*nPtr` is cubed:

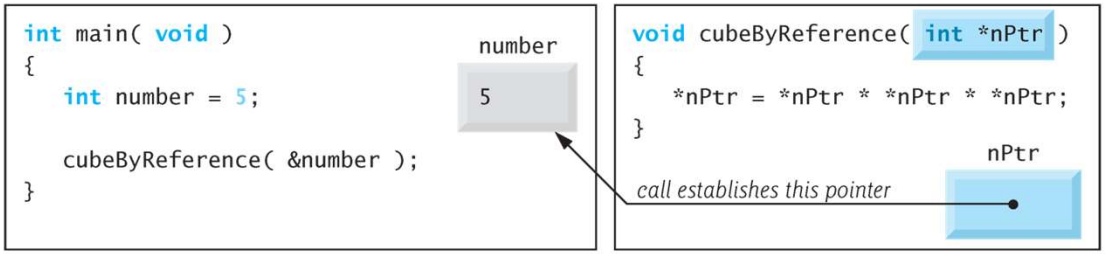


Fig. 7.9 | Analysis of a typical call-by-reference with a pointer argument.

Step 3: After `*nPtr` is cubed and before program control returns to `main`:

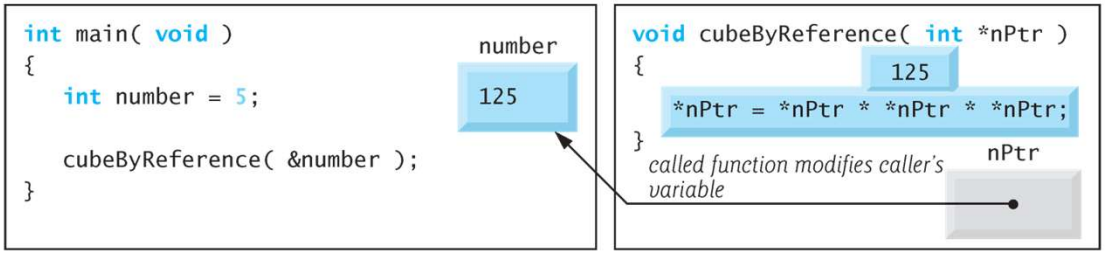


Fig. 7.9 | Analysis of a typical call-by-reference with a pointer argument.

7.4 Passing Arguments to Functions by Reference (Cont.)

- In the function header and in the prototype for a function that expects a single-subscripted array as an argument,

the pointer notation in the parameter list of function `cubeByReference` may be used.

- The compiler does not differentiate between a function that receives a pointer and a function that receives a single-subscripted array.
  - This means that the function must "know" when it's receiving an array or simply a single variable for which it is to perform call by reference.
- When the compiler encounters a function parameter for a single-subscripted array of the form `int b[]`, the compiler converts the parameter to the pointer notation `int *b`.
  - The two forms are interchangeable.

7.5 Using the `const` Qualifier with Pointers

- The `const` qualifier enables you to inform the compiler that the value of a particular variable should not be modified.

Portability Tip 7.1

Although `const` *is well defined in Standard C, some compilers do not enforce it.*

7.5 Using the `const` Qualifier with Pointers (Const.)

- Six possibilities exist for using (or not using) `const` with function parameters—
  - two with call-by-value parameter passing and
  - four with call-by-reference parameter passing.

❑Always award a function enough access to the data in its parameters to accomplish its specified task, **but no more**.

7.5 Using the `const` Qualifier with Pointers (Const.)

- all calls in C are call-by-value
  - a copy of the argument in the function call is made and passed to the function.
  - If the copy is modified in the function, the original value in the caller does not change.

- In many cases, a value passed to a function is modified so the function can accomplish its task.
- However, in some instances, the value should not be altered in the called function, even though it manipulates only a copy of the original value.
  - Consider a function that takes a single-subscripted array and its size as arguments and prints the array. The size of the array is used in the function body to determine the high subscript of the array, so the loop can terminate when the printing is completed. Neither the size of the array nor its contents should change in the function body.

7.5 Using the `const` Qualifier with Pointers (Const.)

- If an attempt is made to modify a value that is declared `const`, the compiler catches it and issues either a warning or an error, depending on the particular compiler.

Software Engineering Observation 7.2

Only one value in a calling function can be altered when using call-by-value. That value must be assigned from the return value of the function to a variable in the caller. To modify multiple variables from a calling function in a called function, use call-by-reference.

7.5 Using the `const` Qualifier with Pointers (Const.)

- There are four ways to pass a pointer to a function:
  1) a non-constant pointer to non-constant data,
  2) a constant pointer to nonconstant data,
  3) a non-constant pointer to constant data, and
  4) a constant pointer to constant data.

7.5 Using the `const` Qualifier with Pointers (Const.)

1

- The highest level of data access is granted by **a non-constant pointer to non-constant data**.
  - the data can be modified through the dereferenced pointer, and
  - the pointer can be modified to point to other data items.

  - A declaration for a non-constant pointer to non-constant data does not include `const`.
  - Such a pointer might be used to receive a string as an argument to a function that uses pointer arithmetic to process (and possibly modify) each character in the string.

```
3     non-constant pointer to non-constant data */
4
5  #include <stdio.h>
6  #include <ctype.h>
7
8  void convertToUppercase( char *sPtr ); /* prototype */
9
10 int main( void )
11 {
12    char string[] = "characters and $32.98"; /* initialize char array */
13
14    printf( "The string before conversion is: %s", string );
15    convertToUppercase( string );
16    printf( "\nThe string after conversion is: %s\n", string );
17    return 0; /* indicates successful termination */
18 } /* end main */
19
20 /* convert string to uppercase letters */
21 void convertToUppercase( char *sPtr )
22 {
23    while ( *sPtr != '\0' ) { /* current character is not '\0' */
24
25       if ( islower( *sPtr ) ) { /* if character is lowercase, */
26          *sPtr = toupper( *sPtr ); /* convert to uppercase */
27       } /* end if */
28
29       ++sPtr; /* move sPtr to the next character */
30    } /* end while */
31 } /* end f
```

```
The string before conversion is: characters and $32.98
The string after conversion is: CHARACTERS AND $32.98
```

processes the array `string` (pointed to by `sPtr`) one character at a time using pointer arithmetic

`islower` tests the character contents of the address pointed to by `sPtr`
`toupper` is called to convert the character to its corresponding uppercase letter

7.5 Using the `const` Qualifier with Pointers (Const.)

2

- A **non-constant pointer to constant data** can be modified to point to any data item of the appropriate type, but the data to which it points cannot be modified.
  - Such a pointer might be used to receive an array argument to a function that will process each element without modifying the data.

- For example,

```
const char * sPtr ;
```

The declaration is read from right to left as "`sPtr` is a pointer to a character constant."

```
2     Printing a string one character at a time using
3     a non-constant pointer to constant data */
4
5  #include <stdio.h>
6
7  void printCharacters( const char *sPtr );
8
9  int main( void )
10 {
11    /* initialize char array */
12    char string[] = "print characters of a string";
13
14    printf( "The string is:\n" );
15    printCharacters( string );
16    printf( "\n" );
17    return 0; /* indicates successful termination */
18 } /* end main */
19
20 /* sPtr cannot modify the character to which it points,
21    i.e., sPtr is a "read-only" pointer */
22 void printCharacters( const char *sPtr )
23 {
24    /* loop through entire string */
25    for ( ; *sPtr != '\0'; sPtr++ ) { /* no initialization */
26       printf( "%c", *sPtr );
27    } /* end for */
28 } /* end function printCharacters */
```

```
The string is:
print characters of a string
```

```
1  /* Fig. 7.12: fig07_12.c
2     Attempting to modify data through a
3     non-constant pointer to constant data. */
4  #include <stdio.h>
5  void f( const int *xPtr ); /* prototype */
6
7
8  int main( void )
9  {
10    int y; /* define y */
11
12    f( &y ); /* f attempts illegal modification */
13    return 0; /* indicates successful termination */
14 } /* end main */
15
16 /* xPtr cannot be used to modify the
17    value of the variable to which it points */
18 void f( const int *xPtr )
19 {
20    *xPtr = 100; /* error: cannot modify a const object */
21 } /* end function f */
```

function attempts to modify the data pointed to by `xPtr` in line 20—which results in a compilation error

```
Compiling...
FIG07_12.c
c:\examples\ch07\fig07_12.c(22) : error C2166: l-value specifies const object
Error executing cl.exe.

FIG07_12.exe - 1 error(s), 0 warning(s)
```

7.5 Using the `const` Qualifier with Pointers (Const.)

3

- A **constant pointer to non-constant data** always points to the same memory location, and the data at that location can be modified through the pointer.
  - This is the default for an array name.
  - An array name is a constant pointer to the beginning of the array.
  - All data in the array can be accessed and changed by using the array name and array subscripting.
- A constant pointer to non-constant data can be used to receive an array as an argument to a function that accesses array elements using only array subscript notation.
- Pointers that are declared `const` must be initialized when they're defined (if the pointer is a function parameter, it's initialized with a pointer that is passed to the function).

```c
/* Fig. 7.13: fig07_13.c
   Attempting to modify a constant pointer to non-constant data */
#include <stdio.h>

int main( void )
{
   int x; /* define x */
   int y; /* define y */

   /* ptr is a constant pointer to an integer that can be modified
      through ptr, but ptr always points to the same memory location */
   int * const ptr = &x;

   *ptr = 7; /* allowed: *ptr is not const */
   ptr = &y; /* error: ptr is const; cannot assign new address */
   return 0; /* indicates successful termination */
} /* end main */
```

The definition is read from right to left as "`ptr` is a constant pointer to an integer."

Fig. 7.13 | Attempting to modify a constant pointer to non-constant data.

```
Compiling...
FIG07_13.c
c:\examples\ch07\FIG07_13.c(15) : error C2166: l-value specifies const object
Error executing cl.exe.

FIG07_13.exe - 1 error(s), 0 warning(s)
```

7.5 Using the `const` Qualifier with Pointers (Const.)

4

- The least access privilege is granted by **a constant pointer to constant data**.
  - Such a pointer always points to the same memory location, and the data at that memory location cannot be modified.
  - This is how an array should be passed to a function that only looks at the array using array subscript notation and does not modify the array.

```c
/* Fig. 7.14: fig07_14.c
   Attempting to modify a constant pointer to constant data. */
#include <stdio.h>

int main( void )
{
   int x = 5; /* initialize x */
   int y; /* define y */

   /* ptr is a constant pointer to a constant integer. ptr always
      points to the same location; the integer at that location
      cannot be modified */
   const int *const ptr = &x;

   printf( "%d\n", *ptr );
   *ptr = 7; /* error: *ptr is const; cannot assign new value */
   ptr = &y; /* error: ptr is const; cannot assign new address */
   return 0; /* indicates successful termination */
} /* end main */
```

is read from right to left as "`ptr` is a constant pointer to an integer constant."

```
Compiling...
FIG07_14.c
c:\examples\ch07\FIG07_14.c(17) : error C2166: l-value specifies const object
c:\examples\ch07\FIG07_14.c(18) : error C2166: l-value specifies const object
Error executing cl.exe.

FIG07_12.exe - 2 error(s), 0 warning(s)
```

7.6 Bubble Sort Using Call-by-Reference

- Let's improve the bubble sort program
  - to use two functions—`bubbleSort` and `swap`.

- Function `bubbleSort` sorts the array.
  - It calls function `swap` to exchange the array elements `array[j]` and `array[j + 1]`
  - Because `bubbleSort` wants `swap` to have access to the array elements to be swapped, `bubbleSort` passes each of these elements call-by-reference to `swap`—the address of each array element is passed explicitly.

7.6 Bubble Sort Using Call-by-Reference (Cont.)

- Although entire arrays are automatically passed by reference, individual array elements are scalars and are ordinarily passed by value.
- Therefore, `bubbleSort` uses the address operator (&) on each of the array elements in the `swap` call (line 51) to effect call-by-reference as follows
  - `swap( &array[ j ], &array[ j + 1 ] );`
  - Function `swap` receives `&array[j]` in pointer variable `element1Ptr` (line 59).

```
2     This program puts values into an array, sorts the values into
3     ascending order, and prints the resulting array. */
4  #include <stdio.h>
5  #define SIZE 10
6
7  void bubbleSort( int * const array, const int size ); /* prototype */
8
9  int main( void )
10 {
11    /* initialize array a */
12    int a[ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
13
14    int i; /* counter */
15
16    printf( "Data items in original order\n" );
17
18    /* loop through array a */
19    for ( i = 0; i < SIZE; i++ ) {
20       printf( "%4d", a[ i ] );
21    } /* end for */
22
23    bubbleSort( a, SIZE ); /* sort the array */
24
25    printf( "\nData items in ascending order\n" );
26
27    /* loop through array a */
28    for ( i = 0; i < SIZE; i++ ) {
29       printf( "%4d", a[ i ] );
30    } /* end for */
31
32    printf( "\n" );
33    return 0; /* indicates successful termination */
34 } /* end main */
35
```

```
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
```

```
36  /* sort an array of integers using bubble sort algorithm */
37  void bubbleSort( int * const array, const int size )
38  {
39     void swap( int *element1Ptr, int *element2Ptr ); /* prototype */
40     int pass; /* pass counter */
41     int j; /* comparison counter */
42
43     /* loop to control passes */
44     for ( pass = 0; pass < size - 1; pass++ ) {
45
46        /* loop to control comparisons during each pass */
47        for ( j = 0; j < size - 1; j++ ) {
48
49           /* swap adjacent elements if they are out of order */
50           if ( array[ j ] > array[ j + 1 ] ) {
51              swap( &array[ j ], &array[ j + 1 ] );
52           } /* end if */
53        } /* end inner for */
54     } /* end outer for */
55  } /* end function bubbleSort */
56
```

```
57  /* swap values at memory locations to which element1Ptr and
58     element2Ptr point */
59  void swap( int *element1Ptr, int *element2Ptr )
60  {
61     int hold = *element1Ptr;
62     *element1Ptr = *element2Ptr;
63     *element2Ptr = hold;
64  } /* end function swap */
```

7.6 Bubble Sort Using Call-by-Reference (Cont.)

- Several features of function `bubbleSort` should be noted:

- The function header (line 37) declares `array` as `int * const array` rather than `int array[]` to indicate that `bubbleSort` receives a single-subscripted array as an argument (again, these notations are interchangeable).

- Parameter `size` is declared `const` to enforce the principle of least privilege.
  - Although parameter `size` receives a copy of a value in `main`, and modifying the copy cannot change the value in `main`, `bubbleSort` does not need to alter `size` to accomplish its task.

7.6 Bubble Sort Using Call-by-Reference (Cont.)

- *Note:*
- *Another common practice is to pass a pointer to the beginning of the array and a pointer to the location just beyond the end of the array.*
- The difference of the two pointers is the length of the array.

7.6 Bubble Sort Using Call-by-Reference (Cont.)

- In the program, the size of the array is explicitly passed to function `bubbleSort`.
- There are two main benefits to this approach
  - software reusability and
  - proper software engineering.

- By defining the function to receive the array size as an argument, we enable the function to be used by any program that sorts single-subscripted integer arrays of any size.

7.6 Bubble Sort Using Call-by-Reference (Cont.)

- We could have stored the array's size in a global variable that is accessible to the entire program.
  - This would be more efficient, because a copy of the size is not made to pass to the function.
  - However, other programs that require an integer array-sorting capability may not have the same global variable, so the function cannot be used in those programs.

Software Engineering Observation 7.5

Global variables usually violate the principle of least privilege and can lead to poor software engineering. Global variables should be used only to represent truly shared resources, such as the time of day.

7.7 `sizeof` Operator

Performance Tip 7.2

`sizeof` is a compile-time operator, so it does not incur any execution-time overhead.

- C provides the special unary operator `sizeof` to determine the size in bytes of an array (or any other data type) during program compilation.
  - When applied to the name of an array, the `sizeof` operator returns the total number of bytes in the array as an integer.
  - (as in following example)
  - Variables of type `float` are normally stored in 4 bytes of memory, and float `array` is defined to have 20 elements. Therefore, there are a total of 80 bytes in `array`.

7.7 `sizeof` Operator (Cont.)

- Type `size_t` is a type defined by the C standard as the integral type (unsigned or unsigned long) of the value returned by operator sizeof.
  - Type `size_t` is defined in header `<stddef.h>` (which is included by several headers, such as `<stdio.h>`).
  - [*Note: If you attempt to compile and receive errors, simply include `<stddef.h>` in your program.*]

```c
/* Fig. 7.16: fig07_16.c
   Applying sizeof to an array name returns
   the number of bytes in the array. */
#include <stdio.h>

size_t getSize( float *ptr ); /* prototype */

int main( void )
{
   float array[ 20 ]; /* create array */

   printf( "The number of bytes in the array is %d"
           "\nThe number of bytes returned by getSize is %d\n",
           sizeof( array ), getSize( array ) );
   return 0; /* indicates successful termination */
} /* end main */

/* return size of ptr */
size_t getSize( float *ptr )
{
   return sizeof( ptr );
} /* end function getSize */
```

```
The number of bytes in the array is 80
The number of bytes returned by getSize is 4
```

Fig. 7.16 | Applying `sizeof` to an array name returns the number of bytes in the array. (Part 1 of 2.)

7.7 `sizeof` Operator (Cont.)

- The number of elements in an array also can be determined with `sizeof`.
- For example,

```
double real[ 22 ];
```

  - Variables of type `double` normally are stored in 8 bytes of memory.
  - Thus, array `real` contains a total of 176 bytes.
- To determine the number of elements in the array, the following expression can be used:
  - `sizeof( real ) / sizeof( real[ 0 ] )`

```c
   Demonstrating the sizeof operator */
#include <stdio.h>

int main( void )
{
   char c;
   short s;
   int i;
   long l;
   float f;
   double d;
   long double ld;
   int array[ 20 ]; /* create array of 20 int elements */
   int *ptr = array; /* create pointer to array */

   printf( "     sizeof c = %d\tsizeof(char)  = %d"
           "\n     sizeof s = %d\tsizeof(short) = %d"
           "\n     sizeof i = %d\tsizeof(int) = %d"
           "\n     sizeof l = %d\tsizeof(long) = %d"
           "\n     sizeof f = %d\tsizeof(float) = %d"
           "\n     sizeof d = %d\tsizeof(double) = %d"
           "\n    sizeof ld = %d\tsizeof(long double) = %d"
           "\n sizeof array = %d"
           "\n   sizeof ptr = %d\n",
           sizeof c, sizeof( char ), sizeof s, sizeof( short ), sizeof i,
           sizeof( int ), sizeof l, sizeof( long ), sizeof f,
           sizeof( float ), sizeof d, sizeof( double ), sizeof ld,
           sizeof( long double ), sizeof array, sizeof ptr );
   return 0; /* indicates successful termination */
} /* end main */
```

```
     sizeof c = 1       sizeof(char)  = 1
     sizeof s = 2       sizeof(short) = 2
     sizeof i = 4       sizeof(int) = 4
     sizeof l = 4       sizeof(long) = 4
     sizeof f = 4       sizeof(float) = 4
     sizeof d = 8       sizeof(double) = 8
    sizeof ld = 8       sizeof(long double) = 8
sizeof array = 80
  sizeof ptr = 4
```

The results could be different between computers.

7.7 `sizeof` Operator (Cont.)

- Operator `sizeof` can be applied to any variable name, type or value (including the value of an expression).
- When applied to a variable name (that is not an array name) or a constant, the number of bytes used to store the specific type of variable or constant is returned.
- The parentheses used with `sizeof` are required if a type name with two words is supplied as its operand (such as `long double` or `unsigned short`).
  - Omitting the parentheses in this case results in a syntax error.
- The parentheses are not required if a variable name or a one-word type name is supplied as its operand, but they can still be included without causing an error.