مبانی برنامه نویسی
به زبان سی

۱۶، ۱۸ و ۲۰ آذر ۱۳۹۹

جلسه چهاردهم، پانزدهم و شانزدهم

ملکی مجد

مباحث این هفته:

- تابع بازگشتی
- آرایه
  - ساختمان داده استاتیک
- فرستادن آرایه به یک تابع
  - آرایه به عنوان پارامتر ورودی تابع
- مرتب سازی اعضای آرایه

5.14 Recursion

- The programs we've discussed are generally structured as functions that call one another in a disciplined, hierarchical manner.
- For some types of problems, it's useful to have **functions call themselves.**

  A recursive function is a function that calls itself either directly or indirectly through another function.

- Recursion is a complex topic discussed at length in upper-level computer science courses.
  - In this section and the next, simple examples of recursion are presented.

5.14 Recursion (Cont.)

- We consider recursion conceptually first, and then examine several programs containing recursive functions.

- A recursive function is called to solve a problem.
- The function actually knows how to solve only the simplest case(s), or so-called base case(s).

5.14 Recursion (Cont.)

- If the function is called with a base case, the function simply returns a result.

- If the function is called with a more **complex** problem, the function divides the problem into two conceptual pieces: a piece that the function knows how to do and a piece that it does not know how to do.
  - To make recursion feasible, the latter piece must resemble the original problem, but be a **slightly simpler or slightly smaller** version.

5.14 Recursion (Cont.)

- Because this new problem **looks like the original problem**, the function launches (**calls) a fresh copy of itself** to go to work on the smaller problem—this is referred to as a recursive call and is also called the recursion step.
- *The recursion step also includes the keyword* `return`,

  because its result will be combined with the portion of the problem the function knew how to solve to form a result that will be passed back to the original caller, possibly `main`.

The recursion step executes while the original call to the function is still open, i.e., it has not yet finished executing.

5.14 Recursion (Cont.)

- The recursion step can result in **many more such recursive calls**, as the function keeps dividing each problem it's called with into two conceptual pieces.
- In order for the recursion to terminate, **each time the function calls itself with a slightly simpler version of the original problem**,
- this sequence of smaller problems must eventually **converge on the base case**.

- At that point, the function recognizes the base case, returns a result to the previous copy of the function, and a sequence of returns ensues all the way up the line until the original call of the function eventually returns the final result to `main`.

5.14 Recursion (Cont.)

- The factorial of a nonnegative integer *n*, written *n!* (and pronounced "*n* factorial"), is the product

$$n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$$

with 1! equal to 1,

and 0! defined to be 1.

- For example, 5! is the product 5 * 4 * 3 * 2 * 1, which is equal to 120.
- The factorial of an integer, `number`, greater than or equal to 0 can be calculated iteratively (nonrecursively) using a `for` statement as follows:

```
factorial = 1;
for ( counter = number; counter >= 1; counter-- )
   factorial *= counter;
```

5.14 Recursion (Cont.)

- A recursive definition of the factorial function is arrived at by observing the following relationship:

  $n! = n \cdot (n - 1)!$

- For example, 5! is clearly equal to 5 * 4! as is shown by the following:

```
5! = 5 · 4 · 3 · 2 · 1
5! = 5 · (4 · 3 · 2 · 1)
5! = 5 · (4!)
```

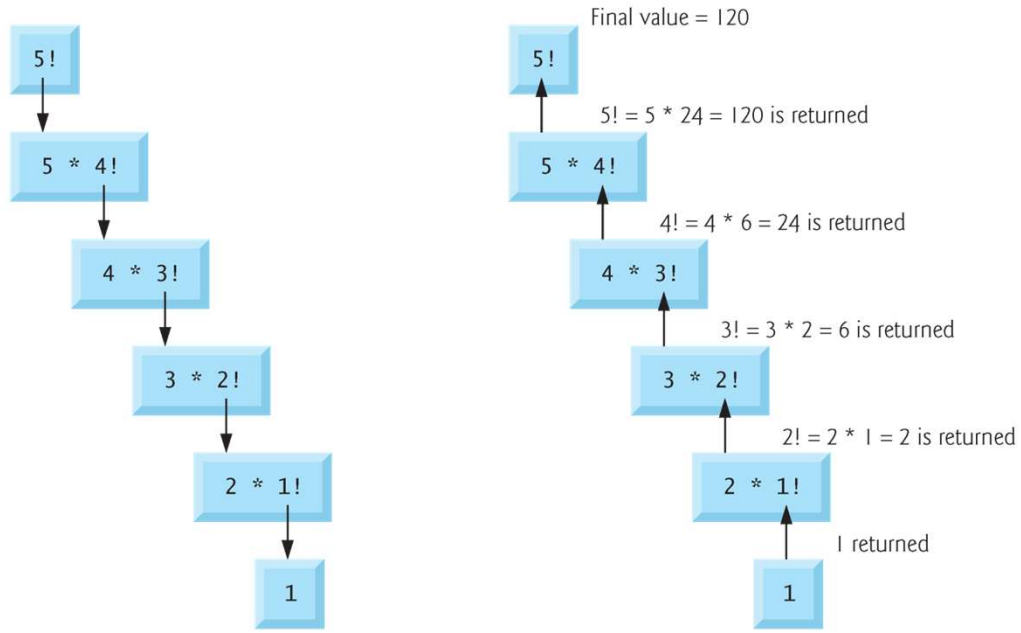
5.14 Recursion (Cont.)

- The recursive `factorial` function first tests whether a terminating condition is true, i.e., whether `number` is less than or equal to 1.

- If `number` is greater than 1, the statement

```
return number * factorial( number - 1 );
```

- The call `factorial( number - 1 )` is a slightly simpler problem than the original calculation `factorial( number )`.



(a) Sequence of recursive calls.

(b) Values returned from each recursive call.

Fig. 5.13 | Recursive evaluation of 5!.

```
3  #include <stdio.h>
4
5  long factorial( long number ); /* function prototype */
6
7  /* function main begins program execution */
8  int main( void )
9  {
10    int i; /* counter */
11
12    /* loop 11 times; during each iteration, calcul
13       factorial( i ) and display result */
14    for ( i = 0; i <= 10; i++ ) {
15       printf( "%2d! = %ld\n", i, factorial( i ) );
16    } /* end for */
17
18    return 0; /* indicates successful termination */
19 } /* end main */
20
21 /* recursive definition of function factorial */
22 long factorial( long number )
23 {
24    /* base case */
25    if ( number <= 1 ) {
26       return 1;
27    } /* end if */
28    else { /* recursive step */
29       return ( number * factorial( number - 1 ) );
30    } /* end else */
31 } /* end function factorial */
```

factorial values become large quickly.
data type `long` has been chosen so the program can calculate factorials greater than 7!

the `factorial` function produces large values so quickly, `double` may ultimately be needed by the user desiring to calculate factorials of larger numbers.

```
 0! = 1
 1! = 1
 2! = 2
 3! = 6
 4! = 24
 5! = 120
 6! = 720
 7! = 5040
 8! = 40320
 9! = 362880
10! = 3628800
```

5.15 Example Using Recursion: Fibonacci Series

- The Fibonacci series
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, …
- begins with 0 and 1 and has the property that each subsequent Fibonacci number is the sum of the previous two Fibonacci numbers.

- The Fibonacci series may be defined recursively as follows:

```
fibonacci(0) = 0
fibonacci(1) = 1
fibonacci(n) = fibonacci(n - 1) + fibonacci(n - 2)
```

```c
#include <stdio.h>

long fibonacci( long n ); /* function prototype */

/* function main begins program execution */
int main( void )
{
   long result; /* fibonacci value */
   long number; /* number input by user */

   /* obtain integer from user */
   printf( "Enter an integer: " );
   scanf( "%ld", &number );

   /* calculate fibonacci value for number input by user */
   result = fibonacci( number );

   /* display result */
   printf( "Fibonacci( %ld ) = %ld\n", number, result );
   return 0; /* indicates successful termination */
} /* end main */

/* Recursive definition of function fibonacci */
long fibonacci( long n )
{
   /* base case */
   if ( n == 0 || n == 1 ) {
      return n;
   } /* end if */
   else { /* recursive step */
      return fibonacci( n - 1 ) + fibonacci( n - 2 );
   } /* end else */
} /* end function fibonacci */
```

```
Enter an integer: 0
Fibonacci( 0 ) = 0
```

```
Enter an integer: 1
Fibonacci( 1 ) = 1
```

```
Enter an integer: 2
Fibonacci( 2 ) = 1
```

```
Enter an integer: 3
Fibonacci( 3 ) = 2
```

```
Enter an integer: 4
Fibonacci( 4 ) = 3
```

```
Enter an integer: 5
Fibonacci( 5 ) = 5
```

```
Enter an integer: 6
Fibonacci( 6 ) = 8
```

```
Enter an integer: 10
Fibonacci( 10 ) = 55
```

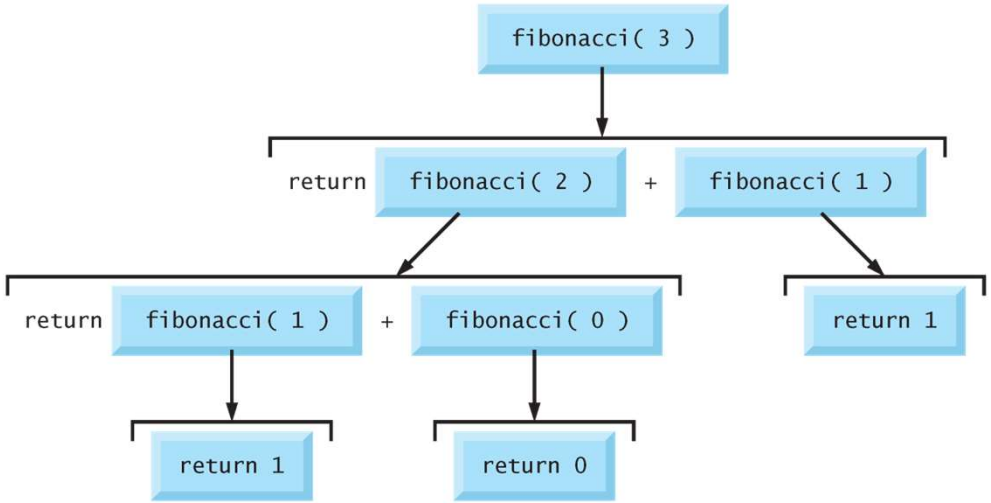
```
Enter an integer: 20
Fibonacci( 20 ) = 6765
```

```
Enter an integer: 30
Fibonacci( 30 ) = 8320
```

```
Enter an integer: 35
Fibonacci( 35 ) = 9227
```

5.15 Example Using Recursion: Fibonacci Series (Cont.)

- If `n` is greater than 1, the recursion step generates two recursive calls, each of which is for a slightly simpler problem than the original call to `fibonacci`.



5.15 Example Using Recursion: Fibonacci Series (Cont.) **exponential complexity**

- Each level of recursion in the `fibonacci` function has a **doubling** effect on the number of calls; i.e., the number of recursive calls that will be executed to calculate the n^{th} Fibonacci number is on **the order of 2^n**.
  - This rapidly gets out of hand.
  - Calculating only the 20^{th} Fibonacci number would require on the order of 2^{20} or about a million calls, calculating the 30^{th} Fibonacci number would require on the order of 2^{30} or about a billion calls, and so on.
- Computer scientists refer to this as **exponential complexity**.
- Problems of this nature humble even the world's most powerful computers!
- Complexity issues in general, and exponential complexity in particular, are discussed in detail in the upper-level computer science curriculum course generally called "Algorithms."

5.15 Example Using Recursion: Fibonacci Series (Cont.)

- Alternate approaches to implementing the recursive Fibonacci algorithm?

5.16 Recursion vs. Iteration

- We studied two functions that can easily be implemented either recursively or iteratively.
- Here, we compare the two approaches and discuss why you might choose one approach over the other in a particular situation.
- Both iteration and recursion are **based on a control structure**: Iteration uses a repetition structure; recursion uses a selection structure.
- Both iteration and recursion **involve repetition**: Iteration explicitly uses a repetition structure; recursion achieves repetition through repeated function calls.

5.16 Recursion vs. Iteration (Cont.)

- Iteration and recursion each involve a **termination test**: Iteration terminates when the loop-continuation condition fails; recursion terminates when a base case is recognized.
- Iteration with counter-controlled repetition and recursion each **gradually approach termination**: Iteration keeps modifying a counter until the counter assumes a value that makes the loop-continuation condition fail; recursion keeps producing simpler versions of the original problem until the base case is reached.

5.16 Recursion vs. Iteration (Cont.)

- Both iteration and recursion **can occur infinitely**: An infinite loop occurs with iteration if the loop-continuation test never becomes false; infinite recursion occurs if the recursion step does not reduce the problem each time in a manner that converges on the base case.
- Recursion has many negatives.
  - It repeatedly invokes the mechanism, and consequently the overhead, of function calls.
  - This can be expensive in both processor time and memory space.
  - Each recursive call causes another copy of the function (can consume considerable memory.)

5.16 Recursion vs. Iteration (Cont.)

- So why choose recursion?

Software Engineering Observation 5.15

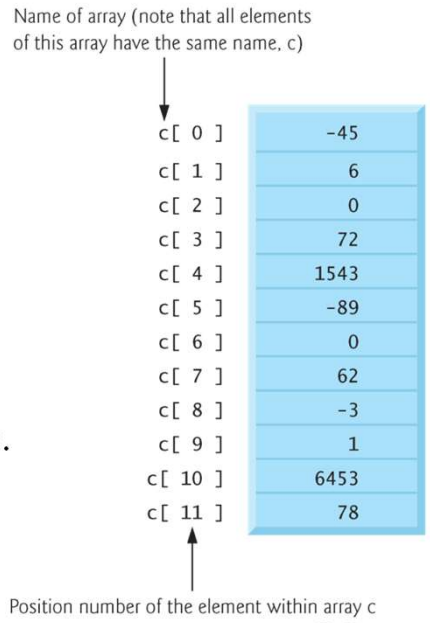
Any problem that can be solved recursively can also be solved iteratively (nonrecursively). A recursive approach is normally chosen in preference to an iterative approach when the recursive approach more naturally mirrors the problem and results in a program that is easier to understand and debug. Another reason to choose a recursive solution is that an iterative solution may not be apparent.

6.2 Arrays

- An array is a group of memory locations related by the fact that they all have the **same name** and the **same type**.
- To refer to a particular location or element in the array, we specify the name of the array and the position number of the particular element in the array.
- Figure : integer array called `c`.
- This array contains 12 elements.
- Any one of these elements may be referred to

  by giving the name of the array followed

  by the position number of the particular element in square brackets (`[]`).



6.2 Arrays (Cont.)

- Array names, like other variable names, can contain only letters, digits and underscores. Array names cannot begin with a digit.
- The position number contained within square brackets is more formally called a subscript (or index).
- A subscript must be an integer or an integer expression.
- For example, if `a = 5` and `b = 6`, then the statement
  - `c[ a + b ] += 2;`

  adds 2 to array element `c[11]`.
- To print the sum of the values contained in the first three elements of array `c`, we'd write
  - `printf( "%d", c[ 0 ] + c[ 1 ] + c[ 2 ] );`

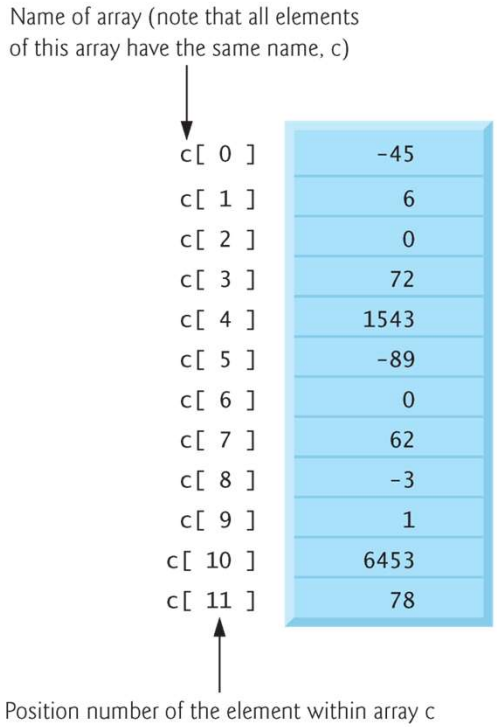


Fig. 6.1 | 12-element array.

Common Programming Error 6.1

It's important to note the difference between the "seventh element of the array" and "array element seven." Because array subscripts begin at 0, the "seventh element of the array" has a subscript of 6, while "array element seven" has a subscript of 7 and is actually the eighth element of the array. This is a source of "off-by-one" errors.

6.2 Arrays (Cont.)

- The brackets used to enclose the subscript of an array are actually considered to be an operator in C.

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

6.3 Defining Arrays

- Arrays occupy space in memory.
- You specify the type of each element and the number of elements required by each array so that the computer may reserve the appropriate amount of memory.
- To tell the computer to reserve 12 elements for integer array `c`, use the definition
    - `int c[ 12 ];`

6.3 Defining Arrays (Cont.)

- The following definition
  - `int b[ 100 ], x[ 27 ];`

  reserves 100 elements for integer array `b` and 27 elements for integer array `x`.

```c
/* Fig. 6.3: fig06_03.c
   initializing an array */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
   int n[ 10 ]; /* n is an array of 10 integers */
   int i; /* counter */

   /* initialize elements of array n to 0 */
   for ( i = 0; i < 10; i++ ) {
      n[ i ] = 0; /* set element at location i to 0 */
   } /* end for */

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

   /* output contents of array n in tabular format */
   for ( i = 0; i < 10; i++ ) {
      printf( "%7d%13d\n", i, n[ i ] );
   } /* end for */

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

```
Element         Value
      0             0
      1             0
      2             0
      3             0
      4             0
      5             0
      6             0
      7             0
      8             0
      9             0
```

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

The elements of an array can also be initialized when the array is defined by following the definition with an equals sign and braces, {}, containing a comma-separated list of initializers.

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

/* function main begins program execution */
int main( void )
{
   /* use initializer list to initialize array n */
   int n[ 10 ] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
   int i; /* counter */

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

   /* output contents of array in tabular format */
   for ( i = 0; i < 10; i++ ) {
      printf( "%7d%13d\n", i, n[ i ] );
   } /* end for */

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

```
Element         Value
      0            32
      1            27
      2            64
      3            18
      4            95
      5            14
      6            90
      7            70
      8            60
      9            37
```

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

6.4 Array Examples (Cont.)

- If there are fewer initializers than elements in the array, the remaining elements are initialized to zero.
- For example, the elements of the array `n` in Fig. 6.3 could have been initialized to zero as follows:
  - `int n[ 10 ] = { 0 };`
- This explicitly initializes the first element to zero and initializes the remaining nine elements to zero because there are fewer initializers than there are elements in the array.
- It's important to remember that arrays are not automatically initialized to zero.

Common Programming Error 6.2

Forgetting to initialize the elements of an array whose elements should be initialized.

6.4 Array Examples (Cont.)

- The array definition
  - `int n[ 5 ] = { 32, 27, 64, 18, 95, 14 };`

  causes a **syntax error** because there are six initializers and only five array elements.

6.4 Array Examples (Cont.)

- If the array size is omitted from a definition with an initializer list, the number of elements in the array will be the number of elements in the initializer list.
- For example,
  - `int n[] = { 1, 2, 3, 4, 5 };`

  would create a five-element array.

6.4 Array Examples (Cont.)

- The `#define` preprocessor directive:

```
#define SIZE 10
```

defines a symbolic constant `SIZE` whose value is 10.

- A symbolic constant is an identifier that is replaced with replacement text by the C preprocessor before the program is compiled.
- When the program is preprocessed, all occurrences of the symbolic constant `SIZE` are replaced with the replacement text 10.
- Using symbolic constants to specify array sizes makes programs more scalable.
  - If the symbolic constant `SIZE` had not been used, we'd have to change the program in several separate places to scale the program to handle 1000 array elements.

```c
/* Fig. 6.5: fig06_05.c
   Initialize the elements of array s to the even integers from 2 to 20 */
#include <stdio.h>
#define SIZE 10 /* maximum size of array */

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

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

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

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

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

```
Element         Value
      0             2
      1             4
      2             6
      3             8
      4            10
      5            12
      6            14
      7            16
      8            18
      9            20
```

6.4 Array Examples (Cont.)

- If the `#define` preprocessor directive in line 4 is terminated with a semicolon, all occurrences of the symbolic constant `SIZE` in the program are replaced with the text `10;` by the preprocessor.
- This may lead to syntax errors at compile time, or logic errors at execution time.

Common Programming Error 6.4

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

sums the values contained in the 12-element integer array a

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

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

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

   printf( "Total of array element values is %d\n", total );
   return 0; /* indicates successful termination */
} /* end main */
```

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

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

6.4 Array Examples (Cont.)

- problem statement:
  - Forty students were asked to rate the quality of the food in the student cafeteria on a scale of 1 to 10 (1 means awful and 10 means excellent).
  - Place the 40 responses in an integer array and summarize the results of the poll.
- We wish to summarize the number of responses of each type (i.e., 1 through 10).

6.4 Array Examples (Cont.)

- The array `responses` (line 17) is a 40-element array of the students' responses.
- We use an 11-element array `frequency` (line 14) to count the number of occurrences of each response.
- We ignore `frequency[0]` because it's logical to have response 1 increment `frequency[1]` rather than `frequency[0]`.
- This allows us to use each response directly as the subscript in the `frequency` array.

```c
/* Fig. 6.7: fig06_07.c
   Student poll program */
#include <stdio.h>
#define RESPONSE_SIZE 40 /* define array sizes */
#define FREQUENCY_SIZE 11

/* function main begins program execution */
int main( void )
{
   int answer; /* counter to loop through 40 responses */
   int rating; /* counter to loop through frequencies 1-10 */

   /* initialize frequency counters to 0 */
   int frequency[ FREQUENCY_SIZE ] = { 0 };

   /* place the survey responses in the responses array */
   int responses[ RESPONSE_SIZE ] = { 1, 2, 6, 4, 8, 5, 9, 7, 8, 10,
        1, 6, 3, 8, 6, 10, 3, 8, 2, 7, 6, 5, 7, 6, 8, 6, 7, 5, 6, 6,
        5, 6, 7, 5, 6, 4, 8, 6, 8, 10 };
   /* for each answer, select value of an element of array responses
      and use that value as subscript in array frequency to
      determine element to increment */
   for ( answer = 0; answer < RESPONSE_SIZE; answer++ ) {
      ++frequency[ responses [ answer ] ];
   } /* end for */

   /* display results */
   printf( "%s%17s\n", "Rating", "Frequency" );

   /* output the frequencies in a tabular format */
   for ( rating = 1; rating < FREQUENCY_SIZE; rating++ ) {
      printf( "%6d%17d\n", rating, frequency[ rating ] );
   } /* end for */

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

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

6.4 Array Examples (Cont.)

- If the data contained invalid values such as 13, the program would attempt to add `1` to `frequency[13]`.
- This would be outside the bounds of the array.
- *C has no array bounds checking to prevent the program from referring to an element that does not exist.*
- Thus, an executing program can "walk off" the end of an array without warning.
- You should ensure that all array references remain within the bounds of the array.

reads numbers from an array and graphs the information in the form of a bar chart or histogram

```c
/* Fig. 6.8: fig06_08.c
   Histogram printing program */
#include <stdio.h>
#define SIZE 10

/* function main begins program execution */
int main( void )
{
   /* use initializer list to initialize array n */
   int n[ SIZE ] = { 19, 3, 15, 7, 11, 9, 13, 5, 17, 1 };
   int i; /* outer for counter for array elements */
   int j; /* inner for counter counts *s in each histogram bar */

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

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

      for ( j = 1; j <= n[ i ]; j++ ) { /* print one bar */
         printf( "%c", '*' );
      } /* end inner for */
      printf( "\n" ); /* end a histogram bar */
   } /* end outer for */

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

```
Element        Value        Histogram
      0           19        *******************
      1            3        ***
      2           15        ***************
      3            7        *******
      4           11        ***********
      5            9        *********
      6           13        *************
      7            5        *****
      8           17        *****************
      9            1        *
```

6.4 Array Examples (Cont.)

- An array version of :
  - Roll a single six-sided die 6000 times to test whether the random number generator actually produces random numbers.

```c
/* Fig. 6.9: fig06_09.c
   Roll a six-sided die 6000 times */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define SIZE 7

/* function main begins program execution */
int main( void )
{
   int face; /* random die value 1 - 6 */
   int roll; /* roll counter 1-6000 */
   int frequency[ SIZE ] = { 0 }; /* clear counts */

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

   /* roll die 6000 times */
   for ( roll = 1; roll <= 6000; roll++ ) {
      face = 1 + rand() % 6;
      ++frequency[ face ]; /* replaces 26-line switch of Fig. 5.8 */
   } /* end for */

   printf( "%s%17s\n", "Face", "Frequency" );

   /* output frequency elements 1-6 in tabular format */
   for ( face = 1; face < SIZE; face++ ) {
      printf( "%4d%17d\n", face, frequency[ face ] );
   } /* end for */

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

```
Face        Frequency
   1             1029
   2              951
   3              987
   4             1033
   5             1010
   6              990
```

6.4 Array Examples (Cont.)
storing strings in character arrays

- So far, the only string-processing capability we have is outputting a string with `printf`.
- A string such as `"hello"` is really a `static` array of individual characters in C.
- A character array can be initialized using a string literal.

```
char string1[] = "first";
```

initializes the elements of array `string1` to the individual characters in the string literal `"first"`.

6.4 Array Examples (Cont.)

- In this case, the size of array `string1` is determined by the compiler based on the length of the string.
- The string `"first"` contains five characters plus a special string-termination character called the null character. Thus, array `string1` actually contains six elements.
  - The character constant representing the null character is `'\0'`.
  - All strings in C end with this character.
- A character array representing a string should always be defined large enough to hold the number of characters in the string and the terminating null character.

6.4 Array Examples (Cont.)

- Character arrays also can be initialized with individual character constants in an initializer list.
- The preceding definition is equivalent to
  - ```
    char string1[] =
       { 'f', 'i', 'r', 's', 't', '\0' };
    ```
- Because a string is really an array of characters, we can access individual characters in a string directly using array subscript notation.
  - For example, `string1[0]` is the character `'f'` and `string1[3]` is the character `'s'`.
    ```

6.4 Array Examples (Cont.)

- We also can input a string directly into a character array from the keyboard using `scanf` and the conversion specifier `%s`.
  - `char string2[ 20 ];`

    creates a character array capable of storing a string of at most 19 characters and a terminating null character.
- The statement `scanf( "%s", string2 );`

  reads a string from the keyboard into `string2`.
- The name of the array is passed to `scanf` without the preceding `&` used with nonstring variables.
  - The `&` is normally used to provide `scanf` with a variable's location in memory so that a value can be stored there.

6.4 Array Examples (Cont.)

- Later when we discuss passing arrays to functions:
  - an array name is the address of the start of the array; therefore, the `&` is not necessary.
- Function `scanf` will read characters until a space, tab, newline or end-of-file indicator is encountered.
- Here:
  - The string should be no longer than 19 characters to leave room for the terminating null character.
  - If the user types 20 or more characters, your program may crash!
  - For this reason, use the conversion specifier `%19s` so that `scanf` does not write characters into memory beyond the end of the array `s`.

6.4 Array Examples (Cont.)

- Function `scanf` reads characters from the keyboard until the first white-space character is encountered—it does not check how large the array is.
  - Thus, `scanf` can write beyond the end of the array.

- It's your responsibility to ensure that the array into which the string is read is capable of holding any string that the user types at the keyboard.

Common Programming Error 6.7

Not providing `scanf` *with a character array large enough to store a string typed at the keyboard can result in destruction of data in a program and other runtime errors. This can also make a system susceptible to worm and virus attacks.*

6.4 Array Examples (Cont.)

- The array `string2` is printed with the statement
  - `printf( "%s\n", string2 );`

- Function `printf`, like `scanf`, does not check how large the character array is. The characters of the string are printed until a terminating null character is encountered.

initializing a character array with a string literal,
reading a string into a character array,
printing a character array as a string and
accessing individual characters of a string.

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

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

   /* read string from user into array string1 */
   printf("Enter a string: ");
   scanf( "%s", string1 ); /* input ended by whitespace character */

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

   /* output characters until null character is reached */
   for ( i = 0; string1[ i ] != '\0'; i++ ) {
      printf( "%c ", string1[ i ] );
   } /* end for */

   printf( "\n" );
   return 0; /* indicates successful termination */
} /* end main */
```

```
Enter a string: Hello there
string1 is: Hello
string2 is: string literal
string1 with spaces between characters is:
H e l l o
```

6.4 Array Examples (Cont.)

- A `static` local variable exists for the duration of the program, but is visible only in the function body.
- We can apply `static` to a local array definition so the array is not created and initialized each time the function is called and the array is not destroyed each time the function is exited in the program.
  - This reduces program execution time, particularly for programs with frequently called functions that contain large arrays.

- Arrays that are `static` are initialized once at compile time.
- If you do not explicitly initialize a `static` array, that array's elements are initialized to zero by the compiler.

```c
/* Fig. 6.11: fig06_11.c
   Static arrays are initialized to zero */
#include <stdio.h>

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

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

   printf( "\n\nSecond call to each function:\n" );
   staticArrayInit();
   automaticArrayInit();
   return 0; /* indicates successful termination */
} /* end main */
```

```
First call to each function:

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

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

Second call to each function:

Values on entering staticArrayInit:
array1[ 0 ] = 5  array1[ 1 ] = 5  array1[ 2 ] = 5
Values on exiting staticArrayInit:
array1[ 0 ] = 10  array1[ 1 ] = 10  array1[ 2 ] = 10

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

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

```c
/* function to demonstrate a static local array */
void staticArrayInit( void )
{
   /* initializes elements to 0 first time function is called */
   static int array1[ 3 ];
   int i; /* counter */

   printf( "\nValues on entering staticArrayInit:\n" );

   /* output contents of array1 */
   for ( i = 0; i <= 2; i++ ) {
      printf( "array1[ %d ] = %d  ", i, array1[ i ] );
   } /* end for */

   printf( "\nValues on exiting staticArrayInit:\n" );

   /* modify and output contents of array1 */
   for ( i = 0; i <= 2; i++ ) {
      printf( "array1[ %d ] = %d  ", i, array1[ i ] += 5 );
   } /* end for */
} /* end function staticArrayInit */
```

```
First call to each function:

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

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

Second call to each function:

Values on entering staticArrayInit:
array1[ 0 ] = 5  array1[ 1 ] = 5  array1[ 2 ] = 5
Values on exiting staticArrayInit:
array1[ 0 ] = 10  array1[ 1 ] = 10  array1[ 2 ] = 10

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

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

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

```
First call to each function:

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

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

Second call to each function:

Values on entering staticArrayInit:
array1[ 0 ] = 5  array1[ 1 ] = 5  array1[ 2 ] = 5
Values on exiting staticArrayInit:
array1[ 0 ] = 10  array1[ 1 ] = 10  array1[ 2 ] = 10

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

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

6.5 Passing Arrays to Functions

- To pass an array argument to a function, specify the name of the array without any brackets.
- For example, if array `hourlyTemperatures` has been defined as
  - `int hourlyTemperatures[ 24 ];`

  the function call
  - `modifyArray( hourlyTemperatures, 24 )`

  passes array `hourlyTemperatures` and its size to function `modifyArray`.

6.5 Passing Arrays to Functions (Cont.)

- Unlike `char` arrays that contain strings, other array types do not have a special terminator.
- For this reason, the **size of an array is passed to the function**, so that the function can process the proper number of elements.

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

6.5 Passing Arrays to Functions (Cont.)

- Next example:

  demonstrates that an array name is really the address of the first element of an array by printing `array`, `&array[0]` and `&array` using the %p conversion specifier

- %p conversion specifier
  - a special conversion specifier for printing addresses.
  - normally outputs addresses as hexadecimal numbers.
  - Hexadecimal (base 16) numbers consist of the digits 0 through 9 and the letters A through F (these letters are the hexadecimal equivalents of the numbers 10–15).

```c
/* Fig. 6.12: fig06_12.c
   The name of an array is the same as &array[ 0 ] */
#include <stdio.h>

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

   printf( "    array = %p\n&array[0] = %p\n   &array = %p\n",
      array, &array[ 0 ], &array );
   return 0; /* indicates successful termination */
} /* end main */
```

```
    array = 0012FF78
&array[0] = 0012FF78
   &array = 0012FF78
```

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

6.5 Passing Arrays to Functions (Cont.)

- Although entire arrays are passed by reference, individual array elements are passed by value exactly as simple variables are.
- To pass an element of an array to a function, use the subscripted name of the array element as an argument in the function call.

6.5 Passing Arrays to Functions (Cont.)

- For a function to receive an array through a function call, the function's parameter list **must specify that an array will be received**.
- For example,

the function header for function `modifyArray` might be written as

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

indicating that `modifyArray` expects to receive an array of integers in parameter `b` and the number of array elements in parameter `size`.

- The size of the array is not required between the array brackets.

the difference between passing an entire array and passing an array element

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

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

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

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

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

```

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

```
24      printf( "\n" );
25
26      /* pass array a to modifyArray by reference */
27      modifyArray( a, SIZE );
28
29      printf( "The values of the modified array are:\n" );
30
31      /* output modified array */
32      for ( i = 0; i < SIZE; i++ ) {
33         printf( "%3d", a[ i ] );
34      } /* end for */
35
36      /* output value of a[ 3 ] */
37      printf( "\n\n\nEffects of passing array element "
38         "by value:\n\nThe value of a[3] is %d\n", a[ 3 ] );
39
40      modifyElement( a[ 3 ] ); /* pass array element a[ 3 ] by value */
41
42      /* output value of a[ 3 ] */
43      printf( "The value of a[ 3 ] is %d\n", a[ 3 ] );
44      return 0; /* indicates successful termination */
45   } /* end main */
46
```

```
Effects of passing entire array by reference:

The values of the original array are:
  0  1  2  3  4
The values of the modified array are:
  0  2  4  6  8


Effects of passing array element by value:

The value of a[3] is 6
Value in modifyElement is 12
The value of a[ 3 ] is 6
```

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

Because arrays are automatically passed by reference, when the called function uses the array name `b`, it will be referring to the array in the caller
(array `hourlyTemperatures` in the preceding call).

```
47 /* in function modifyArray, "b" points to the original array "a"
48    in memory */
49 void modifyArray( int b[], int size )
50 {
51    int j; /* counter */
52
53    /* multiply each array element by 2 */
54    for ( j = 0; j < size; j++ ) {
55       b[ j ] *= 2;
56    } /* end for */
57 } /* end function modifyArray */
58
59 /* in function modifyElement, "e" is a local copy of array element
60    a[ 3 ] passed from main */
61 void modifyElement( int e )
62 {
63    /* multiply parameter by 2 */
64    printf( "Value in modifyElement is %d\n", e *= 2 );
65 } /* end function modifyElement */
```

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

6.5 Passing Arrays to Functions (Cont.)

- There may be situations in your programs in which a function should not be allowed to modify array elements.
- Because arrays are always passed by reference, modification of values in an array is difficult to control.
- C provides the type qualifier `const` to prevent modification of array values in a function.
  - When an array parameter is preceded by the `const` qualifier, the array elements become **constant in the function body**, and any attempt to modify an element of the array in the function body results in a **compile-time error**.
  - This enables you to correct a program so it does not attempt to modify array elements.

```c
/* Fig. 6.14: fig06_14.c
   Demonstrating the const type qualifier with arrays */
#include <stdio.h>

void tryToModifyArray( const int b[] ); /* function prototype */

/* function main begins program execution */
int main( void )
{
   int a[] = { 10, 20, 30 }; /* initialize a */

   tryToModifyArray( a );

   printf("%d %d %d\n", a[ 0 ], a[ 1 ], a[ 2 ] );
   return 0; /* indicates successful termination */
} /* end main */

/* in function tryToModifyArray, array b is const, so it cannot be
   used to modify the original array a in main. */
void tryToModifyArray( const int b[] )
{
   b[ 0 ] /= 2; /* error */
   b[ 1 ] /= 2; /* error */
   b[ 2 ] /= 2; /* error */
} /* end function tryToModifyArray */
```

```
Compiling...
FIG06_14.C
fig06_14.c(22) : error C2166: l-value specifies const object
fig06_14.c(23) : error C2166: l-value specifies const object
fig06_14.c(24) : error C2166: l-value specifies const object
```

Fig. 6.14 | const type qualifier. (Part 1 of 2.)

6.6 Sorting Arrays

- Sorting data (i.e., placing the data into a particular order such as ascending or descending) is one of the most important computing applications.
  - Here, we discuss what is perhaps the simplest known sorting scheme.
- The technique we use is called the bubble sort or the sinking sort because the smaller values gradually "bubble" their way upward to the top of the array like air bubbles rising in water, while the larger values sink to the bottom of the array.
  - The technique is to make several passes through the array.
  - On each pass, successive pairs of elements are compared.
    - If a pair is in increasing order (or if the values are identical), we leave the values as they are.
    - If a pair is in decreasing order, their values are swapped in the array.

```c
   This program sorts an array's values into ascending order */
#include <stdio.h>
#define SIZE 10

/* function main begins program execution */
int main( void )
{
   /* initialize a */
   int a[ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
   int pass; /* passes counter */
   int i; /* comparisons counter */
   int hold; /* temporary location used to swap array elements */

   printf( "Data items in original order\n" );

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

   /* bubble sort */
   /* loop to control number of passes */
   for ( pass = 1; pass < SIZE; pass++ ) {

      /* loop to control number of comparisons per pass */
      for ( i = 0; i < SIZE - 1; i++ ) {

         /* compare adjacent elements and swap them if first
         element is greater than second element */
         if ( a[ i ] > a[ i + 1 ] ) {
            hold = a[ i ];
            a[ i ] = a[ i + 1 ];
            a[ i + 1 ] = hold;
         } /* end if */
      } /* end inner for */
   } /* end outer for */

   printf( "\nData items in ascending order\n" );

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

   printf( "\n" );
   return 0; /* indicates successful termination */
} /* end main */
```

```
17    /* output original array */
18    for ( i = 0; i < SIZE; i++ ) {
19       printf( "%4d", a[ i ] );
20    } /* end for */
21
22    /* bubble sort */
23    /* loop to control number of passes */
24    for ( pass = 1; pass < SIZE; pass++ ) {
25
26       /* loop to control number of comparisons per pass */
27       for ( i = 0; i < SIZE - 1; i++ ) {
28
29          /* compare adjacent elements and swap them if first
30          element is greater than second element */
31          if ( a[ i ] > a[ i + 1 ] ) {
32             hold = a[ i ];
33             a[ i ] = a[ i + 1 ];
34             a[ i + 1 ] = hold;
35          } /* end if */
36       } /* end inner for */
37    } /* end outer for */
38
39    printf( "\nData items in ascending order\n" );
40
41    /* output sorted array */
42    for ( i = 0; i < SIZE; i++ ) {
43       printf( "%4d", a[ i ] );
44    } /* end for */
```

```
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
```

6.6 Sorting Arrays (Cont.)

- On the first pass, the largest value is guaranteed to sink to the bottom element of the array, `a[9]`.
- On the second pass, the second-largest value is guaranteed to sink to `a[8]`.
- On the ninth pass, the ninth-largest value sinks to `a[1]`.
- This leaves the smallest value in `a[0]`, so only nine passes of the array are needed to sort the array, even though there are ten elements.

- Because of the way the successive comparisons are made, a large value may move down the array many positions on a single pass, but a small value may move up only one position.

6.7 Case Study: Computing Mean, Median and Mode Using Arrays

- Following

  array `response` initialized with 99 responses to a survey.
- Each response is a number from 1 to 9.
- The program computes the **mean**, **median** and **mode** of the 99 values.

6.7 Case Study: Computing Mean, Median and Mode Using Arrays (Cont.)

- The **mean** is the arithmetic average of the 99 values.
  - Function `mean` (line 40) computes the mean by totaling the 99 elements and dividing the result by 99.
- The **median** is the "middle value."
  - Function `median` determines the median by calling function `bubbleSort` to sort the array of responses into ascending order, then picking the middle element, `answer[SIZE / 2]`, of the sorted array.
    - When there is an even number of elements, the median should be calculated as the mean of the two middle elements. Function `median` does not currently provide this capability.
  - Function `printArray` (line 156) is called to output the `response` array.
- The **mode** is the value that occurs most frequently among the 99 responses.
  - Function `mode` (line 82) determines the mode by counting the number of responses of each type, then selecting the value with the greatest count.
  - This version of function `mode` does not handle a tie (see Exercise 7.14).
  - Function `mode` also produces a histogram to aid in determining the mode graphically.

```c
/* Fig. 6.16: fig06_16.c
   This program introduces the topic of survey data analysis.
   It computes the mean, median and mode of the data */
#include <stdio.h>
#define SIZE 99

/* function prototypes */
void mean( const int answer[] );
void median( int answer[] );
void mode( int freq[], const int answer[] ) ;
void bubbleSort( int a[] );
void printArray( const int a[] );

/* function main begins program execution */
int main( void )
{
   int frequency[ 10 ] = { 0 }; /* initialize array frequency */

   /* initialize array response */
   int response[ SIZE ] =
      { 6, 7, 8, 9, 8, 7, 8, 9, 8, 9,
        7, 8, 9, 5, 9, 8, 7, 8, 7, 8,
        6, 7, 8, 9, 3, 9, 8, 7, 8, 7,
        7, 8, 9, 8, 9, 8, 9, 7, 8, 9,
        6, 7, 8, 7, 8, 7, 9, 8, 9, 2,
        7, 8, 9, 8, 9, 8, 9, 7, 5, 3,
        5, 6, 7, 2, 5, 3, 9, 4, 6, 4,
        7, 8, 9, 6, 8, 7, 8, 9, 7, 8,
        7, 4, 4, 2, 5, 3, 8, 7, 5, 6,
        4, 5, 6, 1, 6, 5, 7, 8, 7 };

   /* process responses */
   mean( response );
   median( response );
   mode( frequency, response );
   return 0; /* indicates successful termination */
} /* end main */
```

```
39  /* calculate average of all response values */
40  void mean( const int answer[] )
41  {
42     int j; /* counter for totaling array elements */
43     int total = 0; /* variable to hold sum of array elements */
44
45     printf( "%s\n%s\n%s\n", "********", "  Mean", "********" );
46
47     /* total response values */
48     for ( j = 0; j < SIZE; j++ ) {
49        total += answer[ j ];
50     } /* end for */
51
52     printf( "The mean is the average value of the data\n"
53             "items. The mean is equal to the total of\n"
54             "all the data items divided by the number\n"
55             "of data items ( %d ). The mean value for\n"
56             "this run is: %d / %d = %.4f\n\n",
57             SIZE, total, SIZE, (  double  ) total / SIZE );
58  } /* end function mean */
59
```

```
60  /* sort array and determine median element's value */
61  void median( int answer[] )
62  {
63     printf( "\n%s\n%s\n%s\n%s",
64             "********", " Median", "********",
65             "The unsorted array of responses is" );
66
67     printArray( answer ); /* output unsorted array */
68
69     bubbleSort( answer ); /* sort array */
70
71     printf( "\n\nThe sorted array is" );
72     printArray( answer ); /* output sorted array */
73
74     /* display median element */
75     printf( "\n\nThe median is element %d of\n"
76             "the sorted %d element array.\n"
77             "For this run the median is %d\n\n",
78             SIZE / 2, SIZE, answer[ SIZE / 2 ] );
79  } /* end function median */
```

```
81  /* determine most frequent response */
82  void mode( int freq[], const int answer[] )
83  {
84     int rating; /* counter for accessing elements 1-9 of array freq */
85     int j; /* counter for summarizing elements 0-98 of array answer */
86     int h; /* counter for diplaying histograms of elements in array freq */
87     int largest = 0; /* represents largest frequency */
88     int modeValue = 0; /* represents most frequent response */
89
90     printf( "\n%s\n%s\n%s\n",
91             "********", "  Mode", "********" );
93     /* initialize frequencies to 0 */
94     for ( rating = 1; rating <= 9; rating++ ) {
95        freq[ rating ] = 0;
96     } /* end for */
97
98     /* summarize frequencies */
99     for ( j = 0; j < SIZE; j++ ) {
100       ++freq[ answer[ j ] ];
101    } /* end for */
102
103    /* output headers for result columns */
104    printf( "%s%11s%19s\n\n%54s\n%54s\n\n",
105            "Response", "Frequency", "Histogram",
106            "1    1    2    2", "5    0    5    0    5" );
107
108    /* output results */
109    for ( rating = 1; rating <= 9; rating++ ) {
110       printf( "%8d%11d          ", rating, freq[ rating ] );
111
112       /* keep track of mode value and largest frequency value */
113       if ( freq[ rating ] > largest ) {
114          largest = freq[ rating ];
115          modeValue = rating;
116       } /* end if */
117
118       /* output histogram bar representing frequency value */
119       for ( h = 1; h <= freq[ rating ]; h++ ) {
120          printf( "*" );
121       } /* end inner for */
122
123       printf( "\n" ); /* being new line of output */
124    } /* end outer for */
125
126    /* display the mode value */
127    printf( "The mode is the most frequent value.\n"
128            "For this run the mode is %d which occurred"
129            " %d times.\n", modeValue, largest );
130 } /* end function mode */
131
```

```
132 /* function that sorts an array with bubble sort algorithm */
133 void bubbleSort( int a[] )
134 {
135    int pass; /* pass counter */
136    int j; /* comparison counter */
137    int hold; /* temporary location used to swap elements */
138
139    /* loop to control number of passes */
140    for ( pass = 1; pass < SIZE; pass++ ) {
141
142       /* loop to control number of comparisons per pass */
143       for ( j = 0; j < SIZE - 1; j++ ) {
144
145          /* swap elements if out of order */
146          if ( a[ j ] > a[ j + 1 ] ) {
147             hold = a[ j ];
148             a[ j ] = a[ j + 1 ];
149             a[ j + 1 ] = hold;
150          } /* end if */
151       } /* end inner for */
152    } /* end outer for */
153 } /* end function bubbleSort */
154
```

Fig. 6.16 | Survey data analysis program. (Part 7 of 8.)

```
155 /* output array contents (20 values per row) */
156 void printArray( const int a[] )
157 {
158    int j; /* counter */
159
160    /* output array contents */
161    for ( j = 0; j < SIZE; j++ ) {
162
163       if ( j % 20 == 0 ) { /* begin new line every 20 values */
164          printf( "\n" );
165       } /* end if */
166
167       printf( "%2d", a[ j ] );
168    } /* end for */
169 } /* end function printArray */
```

Fig. 6.16 | Survey data analysis program. (Part 8 of 8.)

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


********
 Median
********
The unsorted array of responses is
 6 7 8 9 8 7 8 9 8 9 7 8 9 5 9 8 7 8 7 8
 6 7 8 9 3 9 8 7 8 7 7 8 9 8 9 8 9 7 8 9
 6 7 8 7 8 7 9 8 9 2 7 8 9 8 9 8 9 7 5 3
 5 6 7 2 5 3 9 4 6 4 7 8 9 6 8 7 8 9 7 8
 7 4 4 2 5 3 8 7 5 6 4 5 6 1 6 5 7 8 7

The sorted array is
 1 2 2 2 3 3 3 3 4 4 4 4 4 5 5 5 5 5 5 5
 5 6 6 6 6 6 6 6 6 6 7 7 7 7 7 7 7 7 7 7
 7 7 7 7 7 7 7 7 7 7 7 7 7 8 8 8 8 8 8 8
```

Fig. 6.17 | Sample run for the survey data analysis program. (Part 1 of 2.)

```
 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8
 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9

The median is element 49 of
the sorted 99 element array.
For this run the median is 7

********
  Mode
********
Response  Frequency          Histogram

                                       1    1    2    2
                                  5    0    5    0    5

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

Fig. 6.17 | Sample run for the survey data analysis program. (Part 2 of 2.)