CENG 230

Introduction to C Programming

Week 11– Arrays

Sinan Kalkan

Some slides/content are borrowed from Tansel Dokeroglu, Nihan Kesim Cicekli, and the lecture notes of the textbook by Hanly and Koffman.

Previously on CEng 230!

Function	Description	Example
`sqrt( x )`	square root of *x*	`sqrt( 900.0 )` is 30.0 `sqrt( 9.0 )` is 3.0
`exp( x )`	exponential function e^x	`exp( 1.0 )` is 2.718282 `exp( 2.0 )` is 7.389056
`log( x )`	natural logarithm of *x* (base *e*)	`log( 2.718282 )` is 1.0 `log( 7.389056 )` is 2.0
`log10( x )`	logarithm of *x* (base 10)	`log10( 1.0 )` is 0.0 `log10( 10.0 )` is 1.0 `log10( 100.0 )` is 2.0
`fabs( x )`	absolute value of *x*	`fabs( 13.5 )` is 13.5 `fabs( 0.0 )` is 0.0 `fabs( -13.5 )` is 13.5
`ceil( x )`	rounds *x* to the smallest integer not less than *x*	`ceil( 9.2 )` is 10.0 `ceil( -9.8 )` is -9.0
`floor( x )`	rounds *x* to the largest integer not greater than *x*	`floor( 9.2 )` is 9.0 `floor( -9.8 )` is -10.0
`pow( x, y )`	*x* raised to power *y* (x^y)	`pow( 2, 7 )` is 128.0 `pow( 9, .5 )` is 3.0
`fmod( x, y )`	remainder of *x*/*y* as a floating-point number	`fmod( 13.657, 2.333 )` is 1.992
`sin( x )`	trigonometric sine of *x* (*x* in radians)	`sin( 0.0 )` is 0.0
`cos( x )`	trigonometric cosine of *x* (*x* in radians)	`cos( 0.0 )` is 1.0
`tan( x )`	trigonometric tangent of *x* (*x* in radians)	`tan( 0.0 )` is 0.0

Fig. 5.2 | Commonly used math library functions.

#include<math.h>

Previously on CEng230!

```c
/* Fig. 5.7: fig05_07.c
   Shifted, scaled integers produced by 1 + rand() % 6 */
#include <stdio.h>
#include <stdlib.h>

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

    /* loop 20 times */
    for ( i = 1; i <= 20; i++ ) {

       /* pick random number from 1 to 6 and output it */
       printf( "%10d", 1 + ( rand() % 6 ) );

       /* if counter is divisible by 5, begin new line of output */
       if ( i % 5 == 0 ) {
          printf( "\n" );
       } /* end if */
    } /* end for */

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

```
         6         6         5         5         6
         5         1         1         5         3
         6         6         2         4         2
         6         2         3         4         1
```

Fig. 5.7 | Shifted, scaled random integers produced by 1 + rand() % 6. (Part 2 of 2.)

Previously on CEng 230!

Scope

Previously on CEng 230!

```
/* Fig. 5.12: fig05_12.c
   A scoping example */
#include <stdio.h>

void useLocal( void ); /* function prototype */
void useStaticLocal( void ); /* function prototype */
void useGlobal( void ); /* function prototype */

int x = 1; /* global variable */

/* function main begins program execution */
int main( void )
{
   int x = 5; /* local variable to main */

   printf("local x in outer scope of main is %d\n", x );

   { /* start new scope */
      int x = 7; /* local variable to new scope */

      printf( "local x in inner scope of main is %d\n", x );
   } /* end new scope */

   printf( "local x in outer scope of main is %d\n", x );
```

Fig. 5.12 | Scoping example. (Part 1 of 3.)

Previously on CEng 230!

Scope Rules

- Global scope
  - Identifier defined outside function, known in all functions
  - Used for global variables, function definitions, function prototypes
- Function scope
  - Can only be referenced inside a function body

Scope Rules

Previously on CEng230!

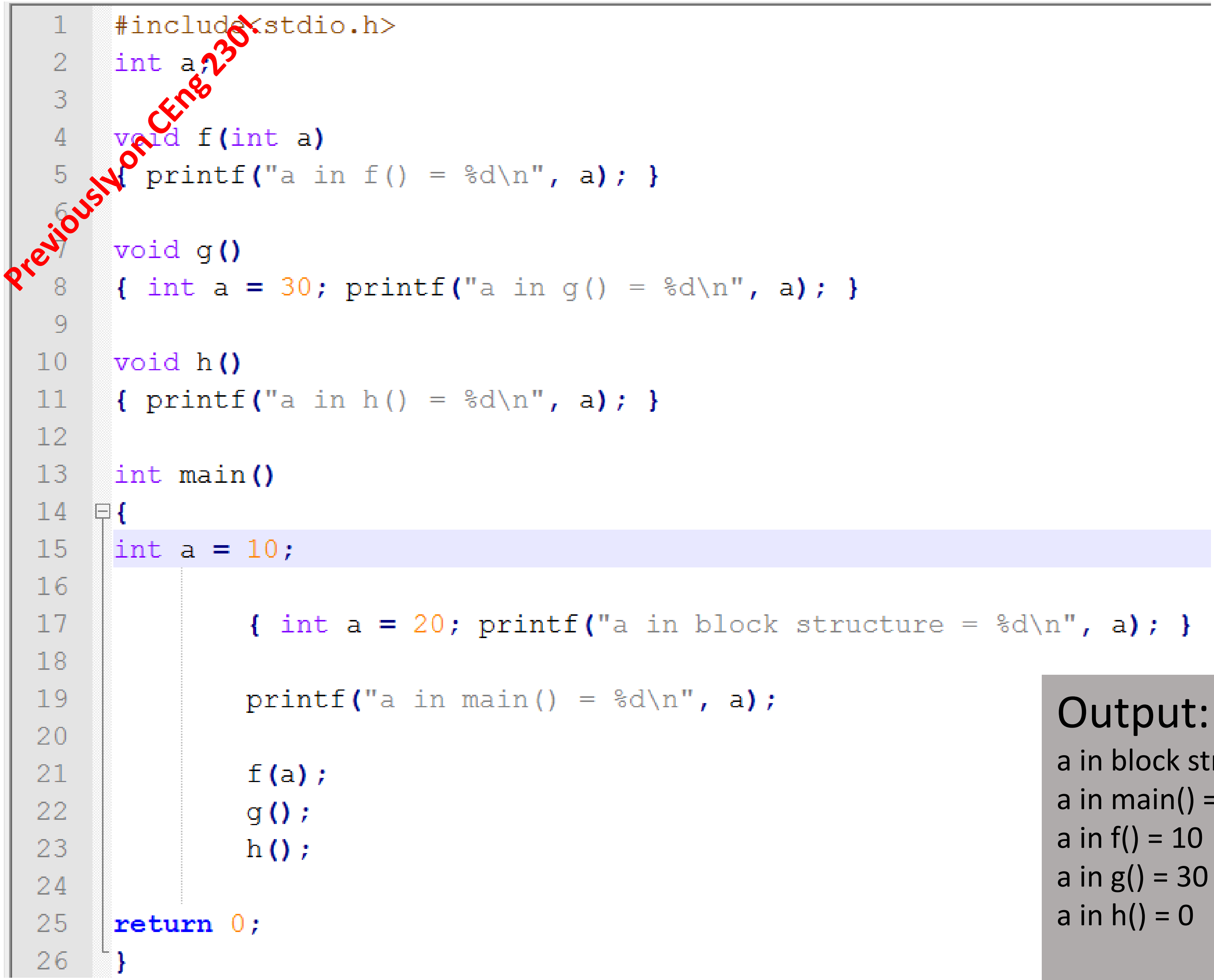
- Block scope
  - Identifier declared inside a block
    - Block scope begins at definition, ends at right brace
  - Used for variables, function parameters (local variables of function)
  - Outer blocks "hidden" from inner blocks if there is a variable with the same name in the inner block
- Function prototype scope
  - Used for identifiers in parameter list

Namespaces

Previously on CEng 230!

- Determines where the definition of variables are valid!
- Global space.
- main() function space.
- Block structures.

Namespace Example



Previously on CEng 230!

```c
#include<stdio.h>
int a;

void f(int a)
{ printf("a in f() = %d\n", a); }

void g()
{ int a = 30; printf("a in g() = %d\n", a); }

void h()
{ printf("a in h() = %d\n", a); }

int main()
{
int a = 10;

        { int a = 20; printf("a in block structure = %d\n", a); }

        printf("a in main() = %d\n", a);

        f(a);
        g();
        h();

return 0;
}
```

Output:

```
a in block structure = 20
a in main() = 10
a in f() = 10
a in g() = 30
a in h() = 0
```

Previously on CEng 230!

Storage-based Types of Variables

Auto vs. register vs. static variables

Storage Classes

Previously on CEng230!

- Storage class specifiers
  - Storage duration – how long an object exists in memory
  - Scope – where object can be referenced in program
  - Linkage – specifies the files in which an identifier is known (more in Chapter 14)
- Automatic storage
  - Object created and destroyed within its block
  - `auto`: default for local variables

    ```
    auto double x, y;
    ```
  - `register`: tries to put variable into high-speed registers
    - Can only be used for automatic variables

    ```
    register int counter = 1;
    ```

Previously on CEng 230!

Storage Classes

- Static storage
  - Variables exist for entire program execution
  - Default value of zero
  - `static`: local variables defined in functions.
    - Keep value after function ends
    - Only known in their own function
  - `extern`: default for global variables and functions
    - Known in any function

Previously on CEng 230!

Parameter passing in functions

Call by Value

Previously on CEng 230!

- The arguments of the function are just copies of the passed data!

```
void f(int a)
{
   a = 10 * a;
}
void g(int b)
{
   b = 10;
   f(b);
   printf("%d",  b);
}
```

Call by Value

Previously on CEng 230!

- So, what do we do? How can I get the changed value?
  - You can use the "return" statement for a variable.
  - If you have more than one variable, you can use global variables.
  - ~~Or, you can use pointers!~~

Today

- Arrays:
  - Storing and working collections of data
  - Declaration & use
  - Initialization
  - Passing arrays to functions
  - Multi-dimensional 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, the definition

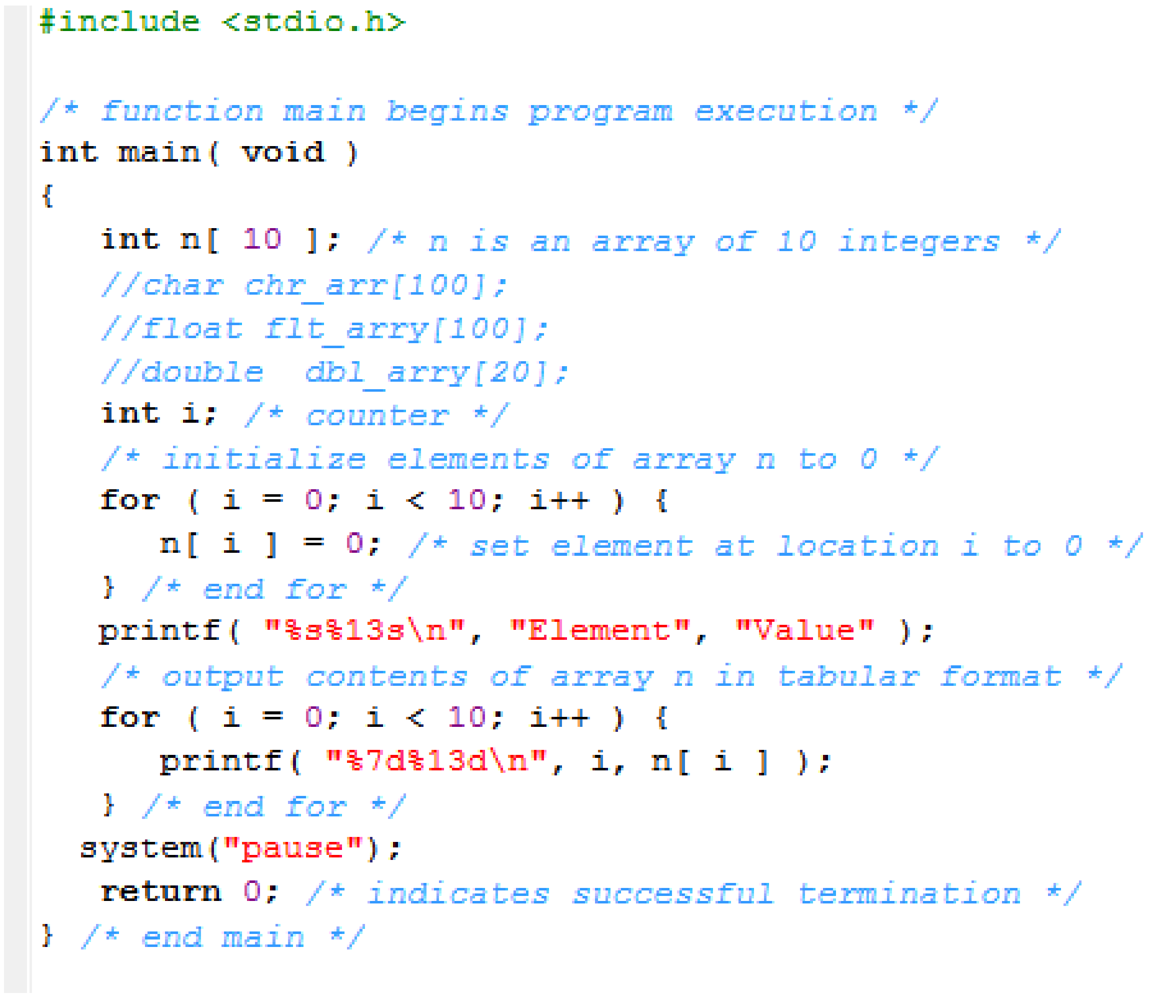
```
int c[ 12 ];
```

is used. The following definition

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

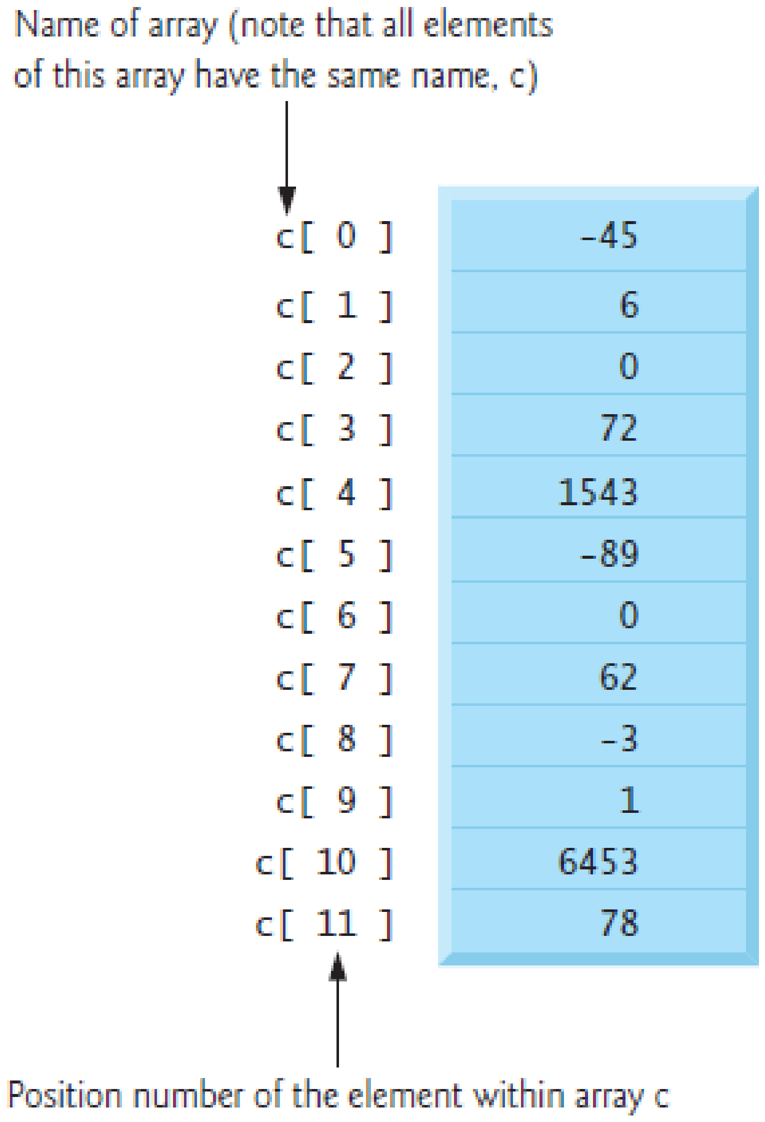
reserves 100 elements for integer array b and 27 elements for integer array x.

```
char    chr_arr[100];
float   flt_arry[100];
double    dbl_arry[20];
```



```c
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
   int n[ 10 ]; /* n is an array of 10 integers */
   //char chr_arr[100];
   //float flt_arry[100];
   //double  dbl_arry[20];
   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 */
  system("pause");
   return 0; /* indicates successful termination */
} /* end main */
```

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.



Name of array (note that all elements of this array have the same name, c)
c[0] -45
c[1] 6
c[2] 0
c[3] 72
c[4] 1543
c[5] -89
c[6] 0
c[7] 62
c[8] -3
c[9] 1
c[10] 6453
c[11] 78
Position number of the element within array c

TABLE 7.1 Statements That Manipulate Array x

Statement	Explanation
`printf("%.1f", x[0]);`	Displays the value of `x[0]`, which is `16.0`.
`x[3] = 25.0;`	Stores the value `25.0` in `x[3]`.
`sum = x[0] + x[1];`	Stores the sum of `x[0]` and `x[1]`, which is `28.0` in the variable `sum`.
`sum += x[2];`	Adds `x[2]` to `sum`. The new `sum` is `34.0`.
`x[3] += 1.0;`	Adds `1.0` to `x[3]`. The new `x[3]` is `26.0`.
`x[2] = x[0] + x[1];`	Stores the sum of `x[0]` and `x[1]` in `x[2]`. The new `x[2]` is `28.0`.

For example, if a = 5 and b = 6, then the statement

```
    c[ a + b ] += 2;
```

adds 2 to array element c[11]. A subscripted array name is an *lvalue*—it can be used on the left side of an assignment.

```
    printf( "%d", c[ 0 ] + c[ 1 ] + c[ 2 ] );
```

```
    x = c[ 6 ] / 2;
```

7.2 Array Subscripts

We use a subscript to differentiate between the individual array elements and to specify which array element is to be manipulated. We can use any expression of type `int` as an array subscript. However, to create a valid reference, the value of this subscript must lie between 0 and one less than the declared size of the array.

EXAMPLE 7.3

Understanding the distinction between an array subscript value and an array element value is essential. The original array `x` from Fig. 7.1 follows. The subscripted variable `x[i]` references a particular element of this array. If `i` has the value `0`, the subscript value is `0`, and `x[0]` is referenced. The value of `x[0]` in this case is `16.0`. If `i` has the value `2`, the subscript value is `2`, and the value of `x[i]` is `6.0`. If `i` has the value `8`, the subscript value is `8`, and we cannot predict the value of `x[i]` because the subscript value is out of the allowable range.

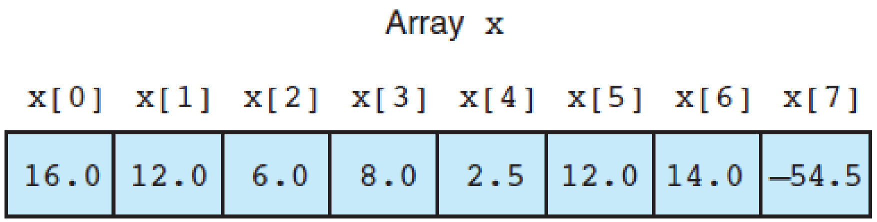


TABLE 7.2 Code Fragment That Manipulates Array x

Statement	Explanation
`i = 5;`	
`printf("%d %.1f", 4, x[4]);`	Displays `4` and `2.5` (value of `x[4]`)
`printf("%d %.1f", i, x[i]);`	Displays `5` and `12.0` (value of `x[5]`)
`printf("%.1f", x[i] + 1);`	Displays `13.0` (value of `x[5]` plus `1`)
`printf("%.1f", x[i] + i);`	Displays `17.0` (value of `x[5]` plus `5`)
`printf("%.1f", x[i + 1]);`	Displays `14.0` (value of `x[6]`)
`printf("%.1f", x[i + i]);`	Invalid. Attempt to display `x[10]`
`printf("%.1f", x[2 * i]);`	Invalid. Attempt to display `x[10]`
`printf("%.1f", x[2 * i - 3]);`	Displays `-54.5` (value of `x[7]`)
`printf("%.1f", x[(int)x[4]]);`	Displays `6.0` (value of `x[2]`)
`printf("%.1f", x[i++]);`	Displays `12.0` (value of `x[5]`); then assigns `6` to `i`
`printf("%.1f", x[--i]);`	Assigns `5` (`6 - 1`) to `i` and then displays `12.0` (value of `x[5]`)
`x[i - 1] = x[i];`	Assigns `12.0` (value of `x[5]`) to `x[4]`
`x[i] = x[i + 1];`	Assigns `14.0` (value of `x[6]`) to `x[5]`
`x[i] - 1 = x[i];`	Illegal assignment statement

Array initialization

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 };
```

The array definition

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

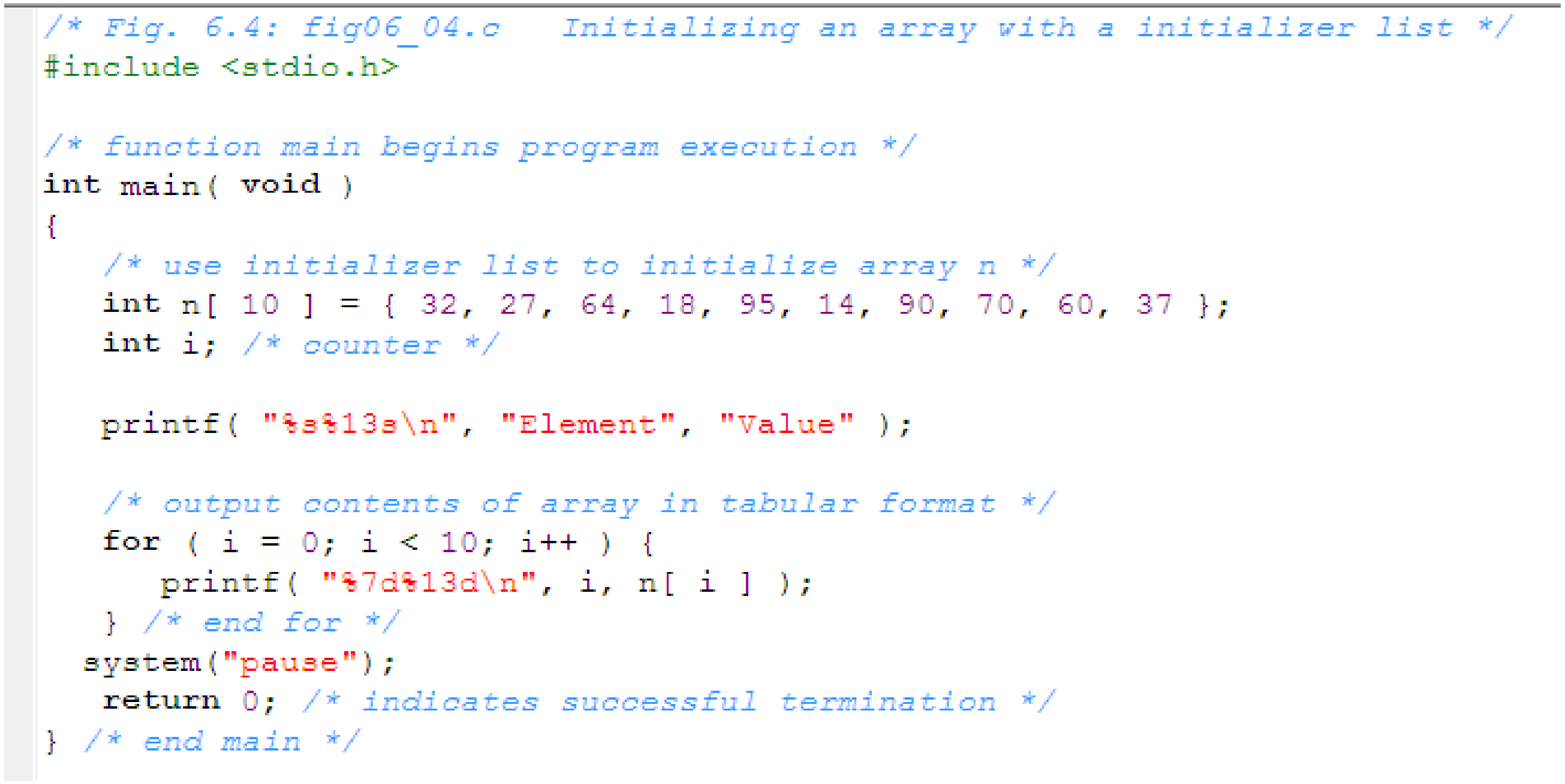
causes a syntax error because there are six initializers and only five array elements.

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,

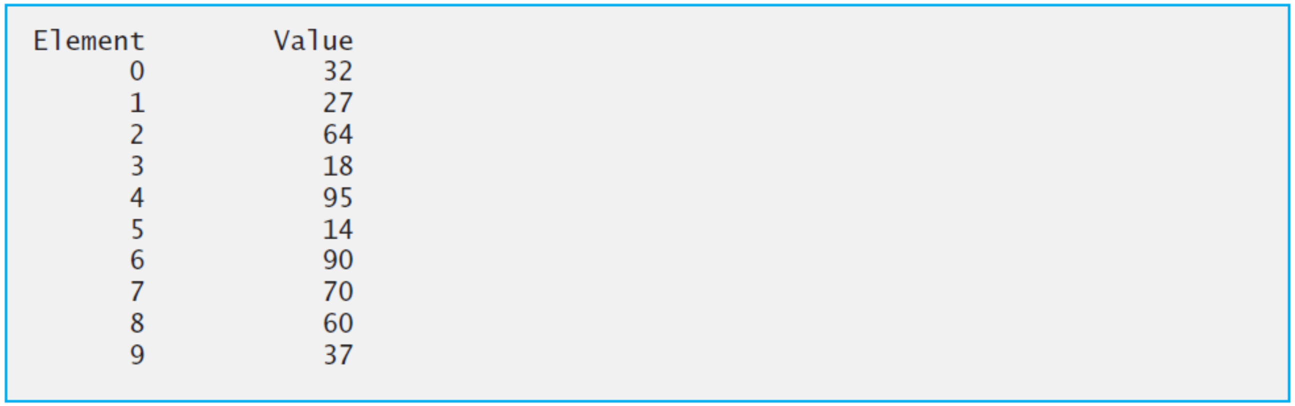
```
int n[] = { 1, 2, 3, 4, 5 };
```

would create a five-element array.

Initializing an Array in a Definition with an initializer List



```c
/* Fig. 6.4: fig06_04.c   Initializing an array with a 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 */
  system("pause");
   return 0; /* indicates successful termination */
} /* end main */
```

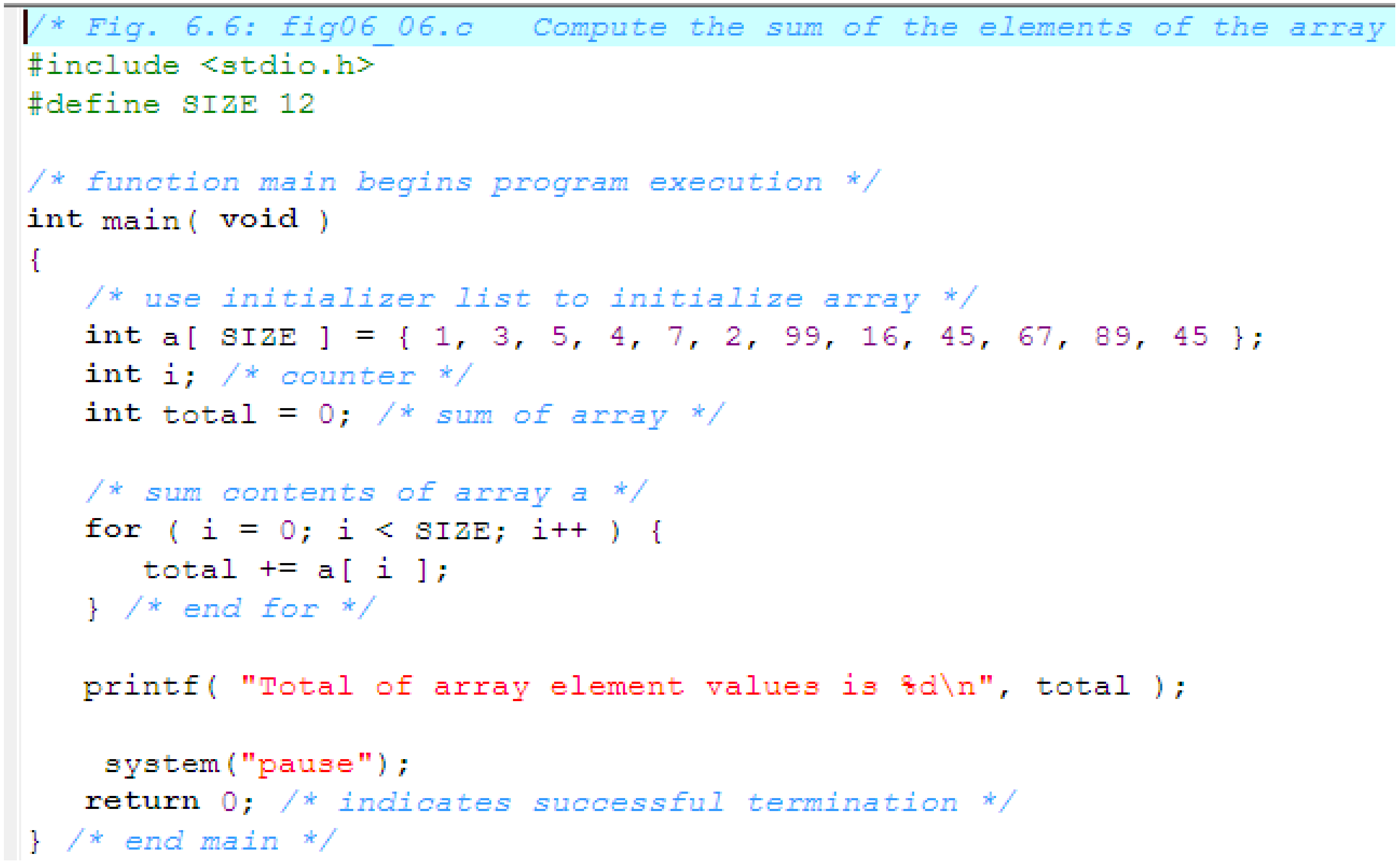


```
Element        Value
      0           32
      1           27
      2           64
      3           18
      4           95
      5           14
      6           90
      7           70
      8           60
      9           37
```

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

```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 */
```

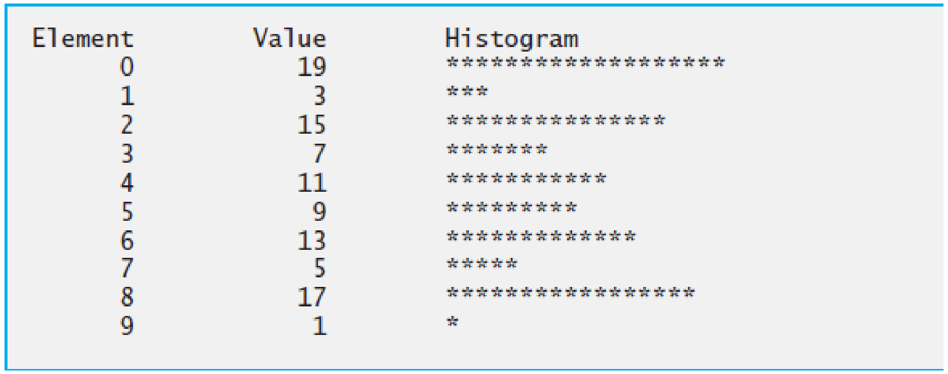
Summing the Elements of an Array



```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 );

    system("pause");
   return 0; /* indicates successful termination */
} /* end main */
```

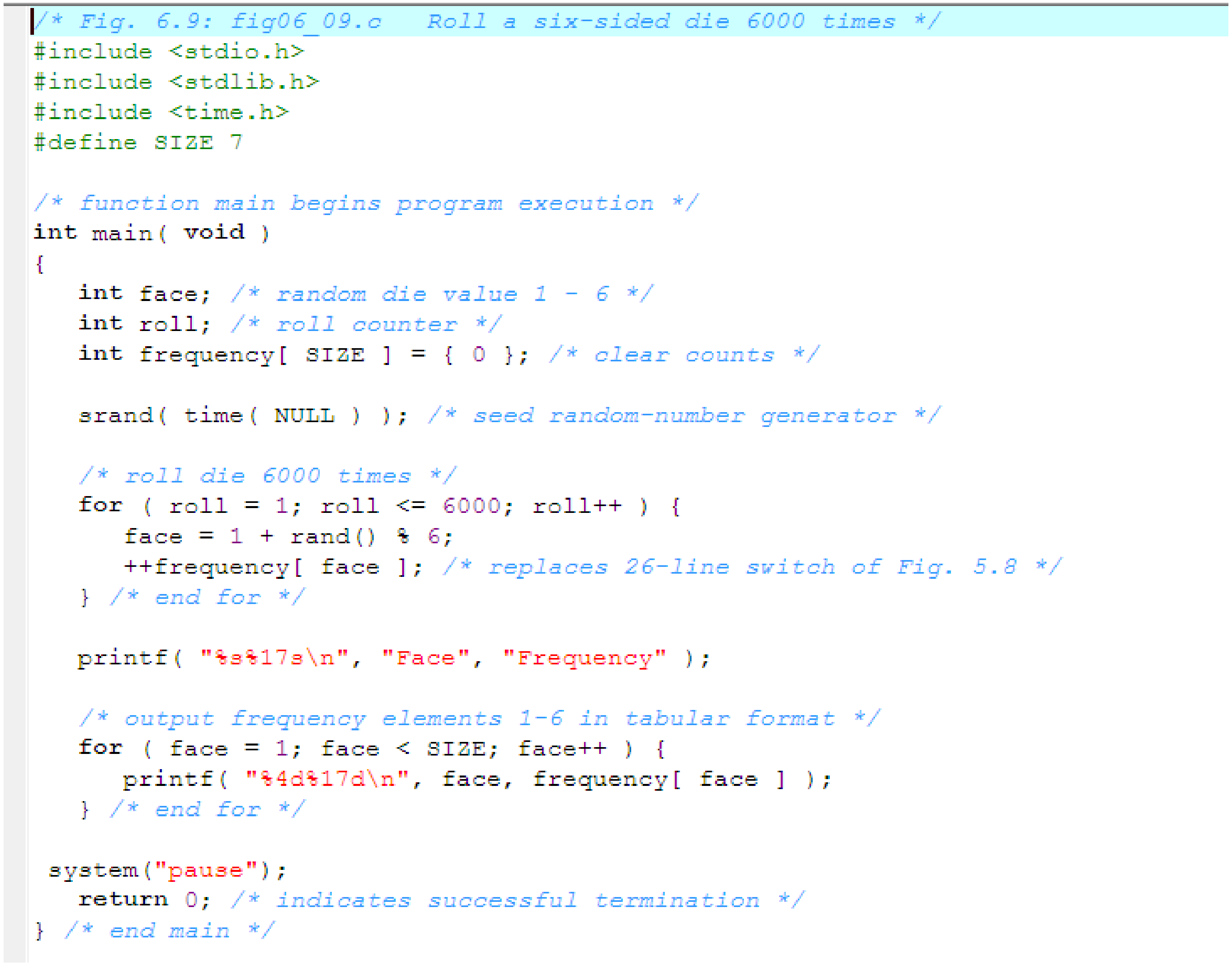
Graphing Array Element Values with Histograms

```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 */

 system("pause");

   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        *
```

Rolling a Die 6000 Times and Summarizing the Results in an Array



```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 */
   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 */

 system("pause");
   return 0; /* indicates successful termination */
} /* end main */
```

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`. 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.

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

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

Fig. 6.13:

```
void modifyArray( int b[], int size )
{
   int j; /* counter */
   /* multiply each array element by 2 */
   for ( j = 0; j < size; j++ ) {
      b[ j ] *= 2;
   } /* end for */
} /* end function modifyArray */

/* in function modifyElement, "e" is a local copy of array element
   a[ 3 ] passed from main */
void modifyElement( int e )
{
   /* multiply parameter by 2 */
   printf( "Value in modifyElement is %d\n", e *= 2 );
} /* end function modifyElement */
```

```
modifyArray( a, SIZE );

modifyElement( a[ 3 ] ); /* pass array element a[ 3 ] by value */
```