Agenda

C Arrays
Arrays

- An array is a group of contiguous memory locations that all have the same type. To refer to a particular location or element in the array, we specify the array's name and the position number of the particular element in the array.

- Figure 1.1 shows an integer array called c, containing 11 elements. Any one of these elements may be referred to by giving the array's name followed by the position number of the particular element. The first element in every array is the zeroth element. An array name, like other variable names, can contain only letters, digits and underscores and cannot begin with a digit.

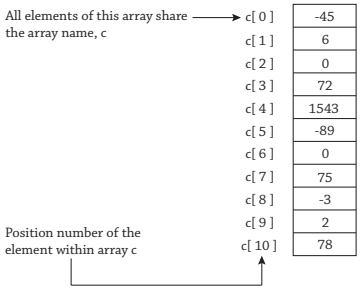

Fig. 1.1 | 11-element array.

C Arrays
Arrays

- The position number within square brackets is called subscript. A subscript must be an integer or an integer expression. For example, if a = 4 and b = 6, the n the statement c[a + b] += 2 adds 2 to array element c[10]. A subscripted array name is an lvalue--it can be used on the left side of an assignment.

- The brackets used to enclose the subscript of an array are actually considered to be an operator in C. They have the same level of precedence as the function call operator. Figure 1.2 shows the precedence and associativity of the operators introduced to this point in the text.

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

Fig 1.2 | Operator precedence and associativity

C Arrays
Defining Arrays

- Arrays occupy space in memory. You specify the type of each array element and the number of elements each array requires so that the computer may reserve the appropriate amount of memory. The following definition reserves 12 elements for integer array c, which has subscripts in the range 0-11.

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

- The definition

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

reserves 100 elements for integer array b and 27 elements for integer array x. These arrays have subscripts in the ranges 0-99 and 0-26, respectively.

- Arrays may contain other data types. For example, an array of type char can store a character string.

Array Examples--Defining an array and Using a Loop to Initialize The Array's Element

- Like any other variables, uninitialized array elements contain garbage values. Figure 1.3 uses for statements to initialize the elements of a 10-element integer array n to zeros and print the array in tabular form.

```
1   // Initializing the elements of an array to zeros.
2   #include <stdio.h>
3
4   // function main begins program execution
5   int main( void )
6   {
7     int n[ 10 ]; // n is an array of 10 integers
8     size_t i; // counter
9
10    // initialize elements of array n to 0
11    for( i = 0; i < 10; ++i ){
12      n[ i ] = 0;
13    } // en for
14
15    printf( "%s%13s\n", "Element", "Value" );
16    // output contents of array n in tabular format
17    for( i = 0; i <10; ++i ){
18      printf( "%7u%13d\n", i, n[ i ] );
19    } // en for
20  } // end main
```

Fig 1.3 | Initializing the elements of an array to zeros.

Array Examples--Defining an array and Using a Loop to Initialize The Array's Element

- Notice that the variable i is declared to be of type size_t, which according to the C standard represents an unsigned integral type. This type is recommeneded for any variable that represents an array's size or an array's subscript. Type size_t is defined in header <stddef.h>, which often included by other headers.

```
1   // Initializing the elements of an array to zeros.
2   #include <stdio.h>
3
4   // function main begins program execution
5   int main( void )
6   {
7     int n[ 10 ]; // n is an array of 10 integers
8     size_t i; // counter
9
10    // initialize elements of array n to 0
11    for( i = 0; i < 10; ++i ){
12      n[ i ] = 0;
13    } // en for
14
15    printf( "%s%13s\n", "Element", "Value" );
16    // output contents of array n in tabular format
17    for( i = 0; i <10; ++i ){
18      printf( "%7u%13d\n", i, n[ i ] );
19    } // en for
20  } // end main
```

Fig 1.3 | Initializing the elements of an array to zeros.

Array Examples--Initializing an Array in Definition with an Initializer List

- 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 array initializer. Figure 1.4 initializes an integer array with 10 values and print the array in tabular form.

```
1   // Initializing the elements of an array to zeros.
2   #include <stdio.h>
3
4   // function main begins program execution
5   int main( void )
6   {
7     // use initializer list to initialize array n
8     int n [ 10 ] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
9     size_t i; // counter
10
11    printf( "%s%13s\n", "Element", "Value" );
12    // output contents of array n in tabular format
13    for( i = 0; i <10; ++i ){
14      printf( "%7u%13d\n", i, n[ i ] );
15    } // en for
16  } // end main
```

Fig 1.4 | Initializing the elements of an array with an initializer list.

Array Examples--Initializing an Array in Definition with an Initializer List

- 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 could have been initialized to zero as follows:

  int n[10] = { 0 }; // initializes entire array to zeros

- 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. You must at least initialize the first element to zero for the remaining elements to be automatically zeroed.

- 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 initialized with the indicated values.

C Arrays
Array Examples--Specifying an Array's Size with a Symbolic Constant

- Figure 1.5 initializes the elements of 10-element array s to the values 2, 4, 6, ..., 20 and prints the array in tabular format. The values are generated by multiplying the loop counter by 2 and adding 2.

```
1   // Initializing the elements of an array to zeros.
2   #include <stdio.h>
3   #define SIZE 10 // maximum size of array
4
5   // function main begins program execution
6   int main( void )
7   {
8     // symbolic constant SIZE can be used to specify array size
9     int s [ SIZE ]; // array s has SIZE elements
10    size_t j; // counter
11
12    for( j = 0; j <10; ++j ){
13      s[ j ] = 2 + 2 * j;
14    } // en for
15
16    printf( "%s%13s\n", "Element", "Value" );
17    // output contents of array n in tabular format
18    for( i = 0; i <10; ++i ){
19      printf( "%7u%13d\n", i, n[ i ] );
20    } // en for
21  } // end main
```

Fig 1.5 | Initialize the elements of array s to the event integers from 2 to 20.

C Arrays
Array Examples--Specifying an Array's Size with a Symbolic Constant

- The #define preprocessor directive is introduced in this program. Line 3 #define SIZE 10 defines a symbolic constant SIZE whose value is 10. A symbolic constant is an identifier that's replaced with replacement text by the C preprocessor before the program is compiled.

- Using symbolic constants to specify array sizes makes programs more scalable. We could have the first loop fill a 1000-element array by simply changing the value of SIZE in the #define directive from 10 to 1000. If the symbolic constant SIZE had not been used, we'd have to change the program in three separate places.

- If the #define preprocessor directive in line 3 is terminated with a semicolon, the preprocessor replaces all occurences of the symbolic constant SIZE in the program with the text 10;. This may lead to syntax errors at compile time, or logic errors at execution time. Remember that the preprocessor is not the C compiler.

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

Common Programming Error
Assigning a value to a symoblic constant in an executable statement is a syntax error. A symbolic constant is not a variable. The compiler does not reserves space for symbolic constasnts as it does for variables that hold values at execution time.

Good Programming Practive
Use only uppercase letters for symbolic constant names. This makes these contants stand out in a program and reminds you that symbolic constants are not variables. In multiword symbolic contants names, separate the words with underscores for readability.

C Arrays
Array Examples--Using Character Arrays to Store and Manipulate Strings

- We've discussed only integer arrays. However, arrays are capable of holding data of any type. We now discuss storing strings in character arrays. So far, only string-processing capability we have is outputting a string with printf.

- Character arrays have several unique features. A character array can be initialized using a string literal. For example,

  char string1[] = "first";

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

- Character arrays also can be initialized with individual character contants in an initializer list. The preceding definition is equivalent to

  char string1[] = { 'f', 'i', 'r', 's', 't', '\0' };

C Arrays
Array Examples--Using Character Arrays to Store and Manipulate Strings

- We also can input a string directly into a character array from the keyboard using scanf and the conversion specifier %s. For example,

  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("%19s", 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 value of an array name is the address of the start of the array; therefore, the & is not necessary.

- A character array representing a string can be output with printf and the %s conversion specifier. The array string2 is printed with the statement

  printf("%s\n", string2);

C Arrays
Array Examples--Using Character Arrays to Store and Manipulate Strings

- Figure 1.6 demonstrates 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.

```
1   // Treating character arrays as strings.
2   #include <stdio.h>
3   #define SIZE 20
4
5   // function main begins program execution
6   int main( void )
7   {
8      char string1[ SIZE ]; //
9      char string2[] = "string literal"; // reserves 15 characters
10     size_t i; // counter
11
12     // read string from user into array string 1
13     printf( "%s", "Enter a string (no longer than 19 characters): " );
14     scanf( "%19s", string1 );
15
16     printf( "string1 is: %s\nstring2 is: %s\n",
               "string1 with spaces between characters is:\n",
               string1, string2 );
17     // output characters until null character is reached
18     for( i = 0; i <SIZE && string1[ i ] != '\0'; ++i ){
19        printf( "%c ", string1[ i ] );
20     } // end for
21     puts( "" );
22  } // end main
```

Fig 1.6 | Treating character arrays as strings.

C Arrays
Array Examples--Static Local Arrays and Local Arrays

- 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.

Performace Tip

In functions that contain automatic arrays where the function is in and out of scope frequently, make the array static so it's not created each time the function is called.

- Arrays that are static are initialized once at program startup. If you do not explicitly initialize a static array, that array's elements are initialied to zero by default.

C Arrays
Array Examples--Static Local Arrays and Local Arrays

- Figure 1.7 demonstrates function staticArrayInit with a local static array and function automaticArrayInit with a local automatic array.

```
1   // Static arrays are initialized to zero if not explicitly initialized.
2   #include <stdio.h>
3
4   void staticArrayInit( void ); // function prototype
5   void automaticArrayInit( void ); // function prototype
6
7   // function main begins program execution
8   int main( void )
9   {
10     puts( "First call to each function:" );
11     staticArrayInit();
12     automaticArrayInit();
13
14     puts( "\n\nSecond call to each function:" );
15     staticArrayInit();
16     automaticArrayInit();
17  } // end main
18  // function to demonstrate a static local array
19  void staticArrayInit( void )
20  {
21     // initializes elements to 0 first time function is called
22     static int array1[ 3 ];
23     size_t i; // counter
24     puts( "\nValues on entering staticArrayInit:" );
25     for( i = 0; i <= 2; ++i ){
```

C Arrays
Array Examples--Static Local Arrays and Local Arrays

```
27        printf( "array1[%u] = %d ", i, array[ i ] );
28     }
29     puts( "\nValues on exiting staticArrayInit:" );
30     for( i = 0; i <= 2; ++i ){
31        printf( "array1[ %u ] = %d ", i, array1[ i ] += 5 );
32     }
33  }// end function staticArrayInit
34
35  // function to demonstrate an automatic local array
36  void automaticArrayInit( void )
37  {
38     int array2[ 3 ] = { 1, 2, 3 };
39     size_t i; // counter
40     puts( "\n\nValues on entering automaticArrayInit:" );
41     for( i = 0; i <=2; ++i ){
42        printf( "array2[ %u ] = %d ", i, array2[ i ] += 5 );
43     }
44     // modify and output contents of array2
45     for( i = 0; i <= 2; ++i ){
46        printf( "array2[ %u ] = %d ", i, array2[ i ] += 5 );
47     }
48  }
```

Fig 1.7 | Static arrays are initialized to zero if not explicitly initialized.

C Arrays
Passing Arrays to Functions

- To pass an array argument to a function, specify the array's name without any brackets. For example, if the array hourlyTemperatures has been defined as

  int hourlyTemperatures [HOURS_IN_A_DAY];

  the function call modifyArray(hourlyTemperatures, HOURS_IN_A_DAY) passes array hourlyTemperatures and its size to function modify array.

- Recall that all arguments in C are passed by value. 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.

Performance Tip
Passing arrays by reference makes sense for performance reasons. If arrays were passed by value, a copy of each element would be passed. For large, frequently passed arrays, this would be time consuming and would consume storage for the copies of the arrays.

C Arrays
Passing Arrays to Functions

- Figure 1.9 demonstrates that an array name is really the address of the first element of the array by printing array, &array[0] using the %p conversion specifier--a special conversion specifier for printing addresses. The %p conversion specifier normally outputs addresses as hexadecimal numbers, but this is compiler independet.

```
1  // Treating character arrays as strings.
2  #include <stdio.h>
3
4  // function main begins program execution
5  int main( void )
6  {
7    char array [ 5 ]; // define an array of size 5
8    printf( "array = %p\n&array[0] = %p\n&array = %p\n", array, &array[ 0 ], &array );
9  } // end main
```

Fig 1.9 | Array name is the same as the address of the array's first element.

C Arrays
Passing Arrays to Functions

- Although entire arrays are passed by reference, individual array elements are passed by value exactly as simple variables are.

- 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. If it's included, the compiler checks that it's greater than zero, then ignores it. Specifying a negative size is a compilation error.

C Arrays
Passing Arrays to Functions--Dif. Between Passing an Entire Array and an Array Element

- Figure 1.10 demonstrates the difference between passing an entire array and passing an array element. The program first prints the five elements of integer array a. Next, a and its size are passed to function modifyArray, where each of a's elements is multiplied by 2.

```
1  // Passing arrays and individual array elements to functions.
2  #include <stdio.h>
3  #define SIZE 5
4
5  void modifyArray( int b[], size_t size );
6  void modifyElement( int e );
7
8  // function main begins program execution
9  int main( void )
10 {
11   int a [ SIZE ] = { 0, 1, 2, 3, 4 }; // initialize array a
12   size_t i; // counter
13   puts( "Effects of passing entire array by reference:\n\nThe "
14         "values of the original array are:" );
15   for( i = 0; i < SIZE; ++i )
16     printf( "%3d, a[ i ] );
17   }
18   puts( "" );
19   modifyArray( a, SIZE );
20   for( i = 0; i < SIZE; ++i )
21     printf( "%3d, a[ i ] );
22   }
```

```
23      printf( "\n\nEffects of passing arry element "
24              "by value:\n\nThe value of a[ 3 ] is %d\n", a[ 3 ] );
25      modifyElement( a[ 3 ] );
26      printf( "The value of a[ 3 ] is %d\n", a[ 3 ] );
27    } // end main
28
29    // in function modifyArray, b points to the original array a in memory
30    void modifyArray( void )
31    {
32      size_t j; // counter
33      // multiply each array element by 2
34      for( j = 0; j < size; ++j )
35        b[ j ] *= 2; // actually modifies original array
36      }
37    }
38    // in function modifyArray, b points to the original array a in memory
39    void modifyArray( void )
40    {
41      printf( "Value in modifyElement is %d\n", e *= 2 );
42    }
```

Fig 1.10 | Passing arrays and individual array elements to function.

- Figure 1.11 demonstrates the const qualifier. Function tryToModifyArray is defined with parameter const int b[], which specifies that array be is constant and cannot be modified. The output shows the error messages produced by the compiler.

```
1   // Using the const type qualifier with arrays.
2   #include <stdio.h>
3    void tryToModifyArray( const int b [] ); // function prototype
4   // function main begins program execution
5   int main( void )
6   {
7     int a[ ] = { 10, 20, 30 }; // initialize array a
8     tryToModifyArray( a );
9     printf( "%d %d %d\n", a[ 0 ], a[ 1 ], a[ 2 ] );
10  } // end main
11  void tryToModifyArray( const int b [] )
12  {
13    b[ 0 ] /= 2;
14    b[ 1 ] /= 2;
15    b[ 2 ] /= 2;
16  } // end main
```

Fig 1.11 | Using the const type qualifier with arrays.

Software Engineering Observation

The const type qualifier can be applied to an array parameter in a function definition to prevent the original array from being modified in the function body. This is another example of the principle of least privilege. A function should not be given the capability to modify an array in the caller unless it's absolutely necessary.

- Sorting data is one of the most important computing applications. Virtually every organization must sort some data, and in many cases massive amounts of it. Sorting data is an intriguing problem which has attracted some of the most intense research efforts in the field of computer science.

- The figure below sorts the values in the elements of the 10-element array a into ascending order. The technique we use is called the bubble sort

```
1   // Passing arrays and individual array elements to functions.
2   #include <stdio.h>
3   #define SIZE 10
4
5   // function main begins program execution
6   int main( void )
7
8     int a [ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 }; // initialize array a
9     int temp; // temporary variable
10    size_t i, j; // comparison counter
11    int hold; // temporary location used to swap array elements
12
13    puts( "Data items in original order" );
14    for( i = 0; i < SIZE; ++i )
15      printf( "%4d, a[ i ] );
16    }
```



```
17    //bubble sort
18    for( i = 1; i < SIZE; ++i ){
19      for( j = 0; j < SIZE - i; ++j ){
20
21        if( a[ j ] > a[ j + 1 ] ){
22          temp = a[ j ];
23          a[ j ] = a[ j + 1 ];
24          a[ j + 1 ] = temp;
25        }
26      }
27    }
28    puts( "Data items in ascending order" );
29    for( i = 0; i < SIZE; ++i ){
30      printf( "%4d, a[ i ] );
31    }
32    puts( "" );
33  } // end main
```

C Arrays
Searching Arrays

- You'll often work with large amounts of data stored in arrays. It may be necessary to determine whether an array contains a value that matches a certain key values. The process of finding a particular element of an array is called searching.
- The linear search compares each element of the array with the search key. Because 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 list.

```
1   // Searching Arrays
2   #include <stdio.h>
3   #define SIZE 100
4
5   size_t linearSearch( const int array [], int key, size_t size );
6
7   // function main begins program execution
8   int main( void ){
9
10    int a [ SIZE ]; // create array a
11    size_t x; // counter for initializing elements 0-99 of array a
12    int searchKey; // value to locate in array a
13    size_t element; // variable to hold location of searchKey or -1
14
15    // create some data
16    for( x = 0; x < SIZE; ++x )
17      a[ x ] = 2 * x;
18    }
19    puts( "Enter integer search key:" );
20    scanf( "%d", &searchKey );
```

C Arrays
Searching Arrays

```
21    element = linearSearch( a, searchKey, SIZE );
22    if( element != -1 ) {
23      printf( "Found value in element %d\n", element );
24    }
25    else{
26      puts( "Value not found" );
27    }
28  } // end main
29
30  size_t linearSearch( const int array [], int key, size_t size ){
31    size_t n; // counter
32    for( n = 0; n < size; ++n ){
33      if( array[ n ] == key ){
34        return n;
35      }
36    }
37    return -1;
38  }
```

C Arrays
Multidimensional Arrays

- Arrays in C can have multiple subscripts. A common use of multiple-subscripted arrays, which the C standard refers to as 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 subscipts: The first identifies the element's row and the second identifies the element's columns.
- Figure below illustrates a double-subscripted array, a. The array contains three rows and four columns, so it's said to be a 3 by 4 array. In general, an array with m rows and n columns is called an m by n array.

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Column Index
Row Index
Array Name

C Arrays
Multidimensional Arrays

- A multidimensional array can be initialized when it's defined much like a single-subscripted array. For example, a double-subscripted array int b[2] [2] could be defined and initialized with;

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

Common Programming Error

Referencing a double-subscripted array element as a[x, y] instead of a[x][y] is a logic error. C interprets a[x, y] as a a[y] (because the comma in this context is treated as a comma operator), so this programmer error is not a syntax error.

C Arrays
Multidimensional Array

- The code below demonstrates defining and initializing double-subscripted arrrays.

```
1   // Initializing multidimensional arrays.
2   #include <stdio.h>
3
4   void printArray( int a [][ 3 ] ); // function prototype
5   int main( void ){
6     int array1 [ 2 ][ 3 ] = { { 1, 2, 3 }, { 4, 5, 6 } };
7     int array2 [ 2 ][ 3 ] = { 1, 2, 3, 4, 5 };
8     int array3 [ 2 ][ 3 ] = { { 1, 2 }, { 4 } };
9     puts( "Values in array1 by row are:" );
10    printArray( array1 );
11    puts( "Values in array2 by rows are:" );
12    printArray( array2 );
13    puts( "Values in array3 by rows are:" );
14    printArray( array3 );
15  } // end main
16  void printArray( int a[][ 3 ] ){
17    size_t i, j; // row and column counters
18    for( i = 0; i <= 1; i ++ ){
19      for( j = 0; j <= 2; j ++ ){
20        printf( "%d ", a[ i ][ j ] );
21      }
22      printf( "\n" );
23    }
24  } // end function printArray
```

C Arrays
Multidimensional Arrays

- When we receive a single-subscripted array as a parameter, the array brackets are empty in the function's parameter list. The first subscript of a multidimensional array is not required either, but all subsequent subscripts are required.

C Arrays
Variable-Length Arrays

- In early versions of C, all arrays had constant size. But what if you don't konw any array's size at compilation time ? To handle this, you'd have to use dynamic memory allocation with malloc and related functions.

- The C standard allows you to handle arrays of unknown size using variable-length arrays (VLAs). A variable-length array is an array whose length, or size, is defined in terms of an expression evaluated at execution time.

C Arrays
Variable-Length Arrays

```
1   // Using variable-length arrays in C99
2   #include <stdio.h>
3   void printArray( int row, int col, int arr[ row ][ col ]; // function prototype
4   int main( void ){
5     int row, col;
6     printf( "%s", "Enter number of rows and columns: " );
7     scanf( "%d%d", &row, &col );
8     int array [ row ][ col ];
9     for( int i = 0; i < row; i ++ ){
10      for( int j = 0; j < col; j ++ ){
11        array[ i ][ j ] = j + i * j
12      } // end for
13    } // end for
14    printArray( row, col, array );
15  } // end main
16
17  void printArray( int row, int col, int arr[ row ][ col ] ){
18    for( int i = 0; i <= 1; i ++ ){
19      for( int j = 0; j <= 2; j ++ ){
20        printf( "%d ", a[ i ][ j ] );
21      }
22      printf( "\n" );
23    }
24  } // end function printArray
```