ENG1008 Programming

- Arrays

Dr Kwee Hiong Lee
kweehiong.lee@singaporetech.edu.sg

Objectives

- To use the array data structure to represent lists and tables
- To define an array, initialise an array and refer to individual elements of an array
- To define symbolic constants
- To pass arrays to functions
- To define and manipulate multidimensional arrays

Introduction

- Arrays are *data structures* consisting of related data items of the same type
- To refer to a *particular location* or any one of these *elements*, we specify or give the *array's* name followed by the position number of the particular element in square brackets ([ ]).
- Figure 6.1 shows an integer array called *c*, containing *12 elements* with array elements c[0], c[1], c[2], c[3], …, c[9], c[10], c[11]
- An array is a group of *contiguous memory locations* that all have the *same type*

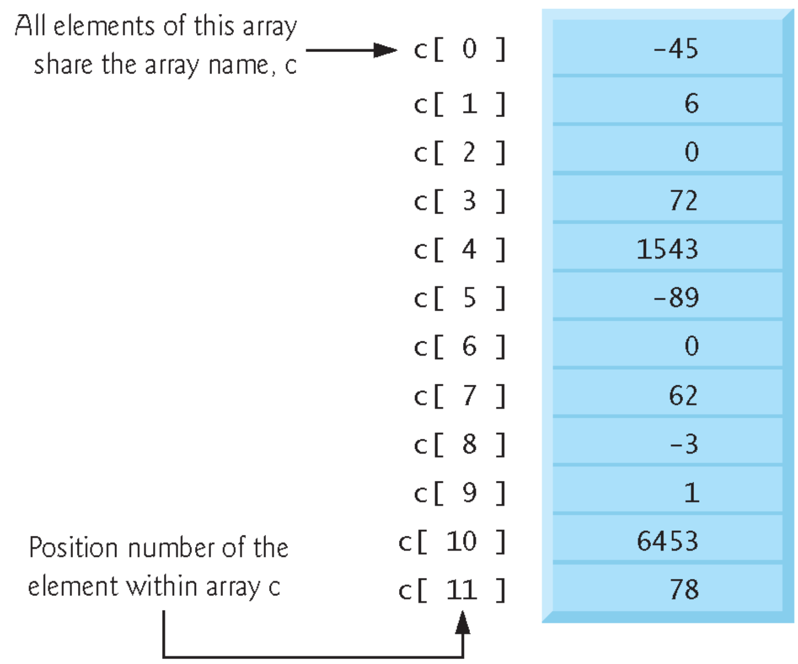

Fig. 6.1 | 12-element array.

One-dimensional array

- Let's start with a one-dimensional array
- The first element in every array is the zeroth element
- An array, like other variable names, can contain only letters, digits and underscores and cannot begin with a digit
- The *position number within square brackets* is 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]

One-dimensional array

- Let’s examine array c (Fig. 6.1) more closely
- The array’s name is **c**
- Its **12** elements are referred to as c[0], c[1], c[2],.., c[10] & c[11]
- The value stored in c[0] is –45, the value of c[1] is 6, c[2] is 0, c[7] is 62 and c[11] is 78
- 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**]);**
- 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 (i.e., the parentheses that are placed after a function name to call that function)

Defining Arrays

- Arrays occupy *space* in memory (*i.e. contiguous 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*
- Note : Character strings and their similarity to arrays

Arrays Initialization

Defining an Array and Using a Loop to Initialize the Array's Elements

- Like any other variables, *uninitialized array elements* contain *garbage values*
- **Figure 6.3** uses for statements to initialize the elements of a *10-element integer array* n to *zeros* and *print the array* in *tabular format*
- The first printf statement (line 16) displays the column heads for the two columns printed in the subsequent for statement
- Notice that the *variable i* is declared to be of **type** size_t (line 9)
  - in C standard represents an **unsigned integral type**
  - This type is recommended for any variable that represents an array's size or an array's subscripts
  - Type size_t is defined in header <stddef.h>, which is often included by other headers (such as <stdio.h>)

Array Examples

```
// Fig. 6.3: fig06_03.c
// Initializing the elements of an array to zeros.
#include <stdio.h>

// function main begins program execution
int main( void )
{
   int n[ 10 ]; // n is an array of 10 integers
   size_t 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( "%7u%13d\n", i, n[ i ] );
   } // end for
} // end main
```

unsigned int

```
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.

Arrays with an Initializer list

SINGAPORE INSTITUTE OF TECHNOLOGY

Initializing an Array in a 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 initializers
- **Figure 6.4** initializes an integer array with 10 values (line 9) and prints the array in tabular format

```
1  // Fig. 6.4: fig06_04.c
2  // Initializing the elements of an array with an initializer list.
3  #include <stdio.h>
4
5  // function main begins program execution
6  int main( void )
7  {
8     // use initializer list to initialize array n
9     int n[ 10 ] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
10    size_t i; // counter
11
12    printf( "%s%13s\n", "Element", "Value" );
13
14    // output contents of array in tabular format
15    for ( i = 0; i < 10; ++i ) {
16       printf( "%7u%13d\n", i, n[ i ] );
17    } // end for
18 } // 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 e

Arrays with an Initializer list

Initializing an Array in a 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 in Fig. 6.3 could have been initialized to zero as follows:

  ```
  // initializes entire array to zeros
  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
- *Arrays* are *not automatically initialized* to *zero*
- *Initialize* at least the *first element* to *zero* for the *remaining elements* to be *automatically zeroed*

Arrays with an Initializer list

Initializing an Array in a Definition with an Initializer List

- Array elements are initialized *before program startup* for *static* arrays and at *runtime* for *automatic* arrays

- 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

Arrays with Symbolic constant

SINGAPORE INSTITUTE OF TECHNOLOGY

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

- **Figure 6.5** initializes the elements of a 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
- The #define preprocessor directive is introduced in this program
- Line 4 SIZE is not a variable
  - #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**
- When the program is *preprocessed*, *all occurrences* of the *symbolic constant SIZE* are *replaced* with the *replacement text 10*

Arrays with Symbolic constant

```c
// Fig. 6.5: fig06_05.c
// Initializing 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
   size_t 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( "%7u%13d\n", j, s[ j ] );
   } // end for
} // end main
```

Fig. 6.5 | Initialize the elements of array s to the even integers from 2 to 20. (Part 1 of 2.)

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

Fig. 6.5 | Initialize the elements to 20. (Part 2 of 2.)

Arrays Examples

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

- Defining the *size* of each array as a *symbolic constant* makes programs more *scalable*
- If the #define preprocessor directive in line 4 is terminated with a semicolon, the preprocessor *replaces all occurrences* of the *symbolic constant SIZE* in the program *with* the *text 10;*
- Error - ending a #define or #include preprocessor directive with a semicolon
- Assigning a value to a symbolic constant in an executable statement is a syntax error
- A *symbolic constant* is *not a variable; the compiler does not reserve space for symbolic constants (vs variables)*
- Use only uppercase letters for *symbolic constant names*

Arrays Examples

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

Fig. 6.8 | Displaying a histogram. (Part 2 of 2.)

```c
// Fig. 6.8: fig06_08.c
// Displaying a histogram.
#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 };
   size_t 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( "%7u%13d        ", i, n[ i ]) ;

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

      puts( "" ); // end a histogram bar
   } // end outer for
} // end main
```

Arrays and Bounds checking

- C has no array bounds checking to prevent the program from referring to an element that does not exist
- Thus, an executing program can *overrun either end* of an *array without warning – a security problem*
- The programmer should ensure that all *array references* remain *within* the *bound* of the *array*
- When looping through an array, the array subscript *should never go below 0* and should always be *less than the total number of elements* in the array (*max size – 1*)

\=
+//

- Passing Arrays to Functions

Passing Arrays to Functions

- To pass a one-dimensional **array argument** to a *function*,
  - specify the array's name without any brackets
- If an array *hourlyTemp* and a symbolic constant is defined as

```
#define HOURS_IN_A_DAY 24
int hourlyTemp[ HOURS_IN_A_DAY ];
```

- The function call to the function *modifyArr*

```
modifyArr( hourlyTemp, HOURS_IN_A_DAY )
```

  passes the array **hourlyTemp** and its *size* to *modifyArr*

- C automatically passes *arrays* to *functions* by reference
- The name of the *array* evaluates to the address *of the* first element of the array *(*i.e. *starting address* of the array*)*.
- Because the **starting address** of the **array** is **passed**, the *called function knows precisely* **where** the **array** is **stored**.

Passing Arrays to Functions

- Therefore, when the *called function* **modifies** *array elements* in its *function body,* it is **actually 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 modifyArr (that we called earlier in this section) might be written as

```
void modifyArr( 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*

Passing Arrays to Functions

SINGAPORE INSTITUTE OF TECHNOLOGY

```
// 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[], size_t size );
void modifyElement( int e );

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

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

   // 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.)

Passing Arrays to Functions

```
   puts( "" );

   // pass array a to modifyArray by reference
   modifyArray( a, SIZE );

   puts( "The values of the modified array are:" );

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

   // output value of a[ 3 ]
   printf( "\n\n\nEffects of passing array element "
      "by value:\n\nThe value of a[3] is %d\n", a[ 3 ] );

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

   // output value of a[ 3 ]
   printf( "The value of a[ 3 ] is %d\n", a[ 3 ] );
} // end main

```

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

Passing Arrays to Functions

```c
// in function modifyArray, "b" points to the original array "a"
// in memory
void modifyArray( int b[], size_t size )
{
   size_t j; // counter

   // multiply each array element by 2
   for ( j = 0; j < size; ++j ) {
      b[ j ] *= 2; // actually modifies original array
   } // 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
```

Fig. 6.13 | Passing arrays and individual array elements

```
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
```

Passing Arrays to Functions

Using the const Qualifier with Array Parameters

- There may be situations in which a function *should not be allowed* to *modify array elements;* it enables you to control a function, so that it does not attempt to modify array elements
- C provides the type qualifier const (for "constant") that can be used 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
- **Figure 6.14** demonstrates the const qualifier
- Function tryToModifyArray (line 19) is defined with parameter const int b[], which specifies that array b is constant and cannot be modified
- The output shows the error messages produced by the compiler — the errors may be different for your compiler

Passing Arrays to Functions

```c
// Fig. 6.14: fig06_14.c
// Using 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 array a

   tryToModifyArray( a );

   printf("%d %d %d\n", a[ 0 ], a[ 1 ], a[ 2 ] );
} // 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
```

```
fig06_14.c(21) : error C2166: l-value specifies const object
fig06_14.c(22) : error C2166: l-value specifies const object
fig06_14.c(23) : error C2166: l-value specifies const object
```

Fig. 6.14 | Using the `const` type qualifier with arrays. (Part 2 of 2.)

- **Multidimensional Arrays**
- **Passing multidimensional arrays to functions**

Multidimensional Arrays

- Arrays in C can be multidimensional arrays that have multiple subscripts
- A *common use* of *multidimensional arrays*, is to *represent* tables of *values* consisting of data arranged in **rows** and **columns**
- *Tables* or *arrays* that require two subscripts to identify a particular element are called double-subscripted arrays (i.e. two-dimensional arrays)
- To identify a particular table element, we must specify **two subscripts**: the first subscript identifies the element's row and the second subscript identifies the element's column
- *Multidimensional arrays* can have *more* than *two subscripts*
- **Figure 6.20** 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*

Multidimensional Arrays

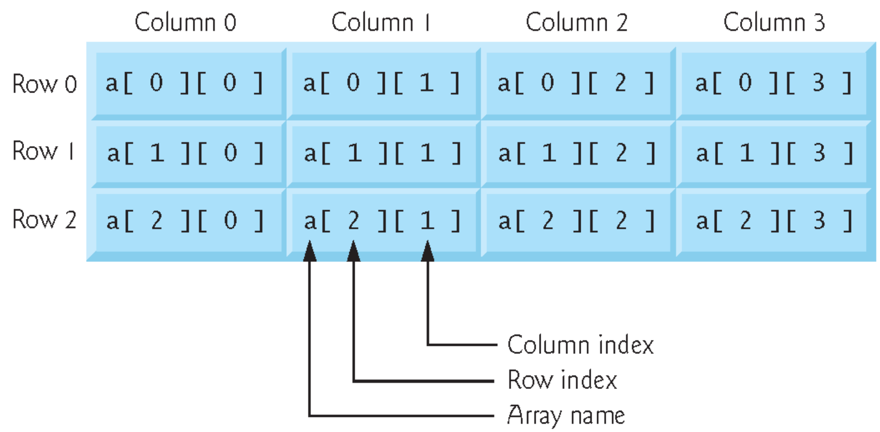

Fig. 6.20 | Double-subscripted array with three rows and four columns.

- Every element in array **a** is identified in **Fig. 6.20** by an element name of the form a[i][j]; **a** is the *name* of the array, and **i** and **j** are the **subscripts/indexs** that *uniquely identify* each element in **a**
- The names of the elements in *row 0* all have a *first subscript of 0*; the names of the elements in *column 3* all have a *second subscript of 3*
- Referencing a double-subscripted array element as a[x,y] instead of a[x][y] is a logic error (this is not a syntax error)

Multidimensional Arrays

SINGAPORE INSTITUTE OF TECHNOLOGY

- 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

row 0 row 1

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

	col 0	col 1
row 0	1	2
row 1	3	4

- The *values* are grouped by row in braces
  - The values in the *first set* of braces *initialize row 0* and the values in the *second set* of braces *initialize row 1*
  - So, the values 1 and 2 initialize elements b[0][0] and b[0][1], respectively, and the values 3 and 4 initialize elements b[1][0] and b[1][1], respectively

- If there are not enough *initializers* for a given row, the remaining elements of that row are initialized to 0
  - Thus,

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

1	0
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

Multidimensional Arrays

- In a *double-subscripted* array, *each row* is basically a *single-subscripted array*
- To locate an element in a particular **row**, the compiler must know *how many elements are in each row* so that it can **skip** the proper number of memory locations when accessing the array
- Thus, when accessing a[1][2] in our example,
  - the compiler knows to skip the four elements of the first row to get to the second row (row 1)
  - then, the compiler accesses element 2 of that row
- Many *common array manipulations* use for repetition statements

Multidimensional Arrays

- For example, the following statement sets all the elements in row 2 of array **a** in Fig. 6.20 to zero:

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

- We specified row 2, so the first subscript is always 2
- The loop varies only the second subscript
- The preceding for statement is equivalent to the assignment statements:

```
a[2][0] = 0;
a[2][1] = 0;
a[2][2] = 0;
a[2][3] = 0;
```

Multidimensional Arrays

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

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

- The **for** statement totals the elements of the array **one row at a time**
- The **outer for** statement begins by setting **row** (i.e., the row subscript) to **0** so that the elements of that row may be totaled by the **inner for** statement
- The *outer for* statement then *increments* **row** *to* **1**, so the elements of that row can be totaled and the *same for* **row** = **2**
- When the **nested for** statement terminates, **total** contains the **sum** of all the elements in the array **a**

Bounds checking for Array Subscripts

- It's important to ensure that *every subscript* you use to *access an array* element is within the array's bounds - that is, *greater than or equal to 0 and less than the number of array elements*
- A two-dimensional array's *row and column subscripts* must be greater than or equal to 0 and less than the numbers of rows and columns, respectively
- Allowing programs to read from or write to array elements *outside the bounds* of arrays are common *security flaws*
- *Reading* from *out-of-bounds array elements* can cause a *program* to *crash* or even appear to *execute correctly* while using *bad data*
- *Writing* to an *out-of-bounds element (known as a buffer overflow)* can *corrupt* a *program's data in memory, crash* a program and allow *attackers* to *exploit* the *system* and execute their own code
- As we stated in the chapter, C provides no automatic bounds checking for arrays, so you must provide your own

Multidimensional Arrays

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

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

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

   puts( "Values in array1 by row are:" );
   printArray( array1 );

   puts( "Values in array2 by row are:" );
   printArray( array2 );

   puts( "Values in array3 by row are:" );
   printArray( array3 );
} // end main
```

Multidimensional Arrays

```c

// function to output array with two rows and three columns
void printArray( int a[][ 3 ] )
{
   size_t i; // row counter
   size_t j; // column counter

   // loop through rows
   for ( i = 0; i <= 1; ++i ) {

      // output column values
      for ( j = 0; j <= 2; ++j ) {
         printf( "%d ", a[ i ][ j ] );
      } // end inner for

      printf( "\n" ); // start new line of output
   } // end outer for
} // end function printArray
```

Fig. 6.21 | Initializing multidimensional arrays. (Part 2 of 3.)

```
Values in array1 by row are:
1 2 3
4 5 6
Values in array2 by row are:
1 2 3
4 5 0
Values in array3 by row are:
1 2 0
4 0 0
```

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

Multidimensional Arrays

- The program defines *three arrays* of *two rows* and *three columns* (six elements each)
- The definition of **array1** (line 11) provides six initializers in two **sublists**
  - The *first* sublist initializes *row 0* of the array to the values 1, 2 and 3; and the *second* sublist initializes *row 1* of the array to the values 4, 5 and 6
  - If the braces around each sublist are removed from the array1 initializer list, the compiler initializes the *elements* of the *first row followed by* the *elements* of the *second row*
- The definition of **array2** (line 12) provides five initializers
  - The *initializers* are *assigned* to the *first row*, *then* the *second row*
  - Any elements that *do not have an explicit initializer* are *initialized to zero automatically*, array2[1][2] is initialized to 0

Multidimensional Arrays

- The definition of **array3** (line 13) provides three initializers in two **sublists**
  - The *sublist* for the first row *explicitly* initializes the first two elements of the first row to 1 and 2
  - The third element is initialized to *zero*
  - The *sublist* for the second row *explicitly* initializes the first element to 4
  - The last two elements are initialized to *zero*
- The program calls **printArray** (lines 26–41) to output each array's elements
- The function definition specifies the array parameter as int a[][3].

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
- The compiler uses these *subscripts* to determine *the locations in memory* of elements in multidimensional arrays
- All array elements are stored consecutively in *memory* regardless of the number of subscripts
- *In a* double-subscripted *array,* the *first row is stored in memory followed by the second row*
- Providing the *subscript values* in a *parameter declaration* enables the compiler to tell the function how to *locate an element* in the *array*

- **Character Arrays**
- **Sorting Arrays**

Character Arrays

Using Character Arrays to Store and Manipulate Strings

- We now discuss storing ***strings*** in *character arrays*
- A string such as "hello" is really an array of individual characters in C
- 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*

Character Arrays

Using Character Arrays to Store and Manipulate Strings

- The character constant representing the **null character** is **‘\0’**
- All strings in C end with this null 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*
- 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'
- We also can input a string directly into a character array from the keyboard using *scanf* and the conversion specifier %s

Character Arrays

Using Character Arrays to Store and Manipulate Strings

- For example,

```c
char string2[20];
```

- creates a character array capable of storing a string of *at most 19 characters* and a *terminating null character*
- The statement

```c
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 & is normally used to provide scanf with a *variable's location in memory* so that a value can be stored there
- As the value of an **array name** is the ***starting address*** of the ***array***; therefore, the *& is not necessary*

Character Arrays

Using Character Arrays to Store and Manipulate Strings

- Function *scanf* will read characters until a *space*, *tab*, *newline* or *end-of-file indicator* is encountered
- The string2 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 or create a security vulnerability
- For this reason, we used the conversion specifier **%19s** so that *scanf reads a maximum of 19 characters* and does not write characters into memory beyond the end of the array string2
- 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

Character Arrays

Using Character Arrays to Store and Manipulate Strings

- Function *scanf* does *not* check how large the array is
- Thus, *scanf* can *write beyond* the *end* of the *array*
- 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 );
```

- 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**.

Sorting Arrays

Sorting Arrays

- **Figure 6.15** sorts the values in the elements of the 10-element array a (line 10) into ascending order
- 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

Sorting Arrays

```
// Fig. 6.15: fig06_15.c
// Sorting 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
   size_t i; // comparisons counter
   int hold; // temporary location used to swap array elements

   puts( "Data items in original order" );

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

```

Fig. 6.15 | Sorting an array's values into ascending order. (Part 1 of 3.)

Sorting Arrays

```
    // 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

    puts( "\nData items in ascending order" );

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

    puts( "" );
 } // end main
```

```
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
```

Fig. 6.15 | Sorting an array's values into ascending order. (Part 3 of 3.)

❖ **Variable-Length Arrays**

Variable-Length Arrays

- In early versions of C, *all arrays* had constant size
- But what if you don't know an 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)
- These are *not arrays whose size can change* - that would compromise the integrity of nearby locations in memory
- A variable-length array is an *array* whose length, or size, is defined in terms of an *expression* evaluated at execution time
- The program **of Fig. 6.23** declares and prints several VLAs.
- Note: This feature is not supported in Microsoft Visual C++.

Variable-Length Arrays

SINGAPORE INSTITUTE OF TECHNOLOGY

```
// Fig. 6.23: figG_14.c
// Using variable-length arrays in C99
#include <stdio.h>

// function prototypes
void print1DArray( int size, int arr[      ] );
void print2DArray( int row, int col, int arr[     ][ col ] );

int main( void )
{
   int arraySize; // size of 1-D array
   int row1, col1, row2, col2; // number of rows and columns in 2-D arrays

   printf( "%s", "Enter size of a one-dimensional array: " );
   scanf( "%d", &arraySize );

   printf( "%s", "Enter number of rows and columns in a 2-D array: " );
   scanf( "%d %d", &row1, &col1 );

   printf( "%s",
      "Enter number of rows and columns in another 2-D array: " );
   scanf( "%d %d", &row2, &col2 );

```

Fig. 6.23 | Using variable-length arrays in C99. (Part 1 of 5.)

Variable-Length Arrays

```
int array[ arraySize ]; // declare 1-D variable-length array
int array2D1[ row1 ][ col1 ]; // declare 2-D variable-length array
int array2D2[ row2 ][ col2 ]; // declare 2-D variable-length array

// test sizeof operator on VLA
printf( "\nsizeof(array) yields array size of %d bytes\n",
   sizeof( array ) );

// assign elements of 1-D VLA
for ( int i = 0; i < arraySize; ++i ) {
   array[ i ] = i * i;
} // end for

// assign elements of first 2-D VLA
for ( int i = 0; i < row1; ++i ) {
   for ( int j = 0; j < col1; ++j ) {
      array2D1[ i ][ j ] = i + j;
   } // end for
} // end for

// assign elements of second 2-D VLA
for ( int i = 0; i < row2; ++i ) {
   for ( int j = 0; j < col2; ++j ) {
      array2D2[ i ][ j ] = i + j;
   } // end for
} // end for
```

Fig. 6.23 | Using variable-length arrays in C99. (Part 2 of 5.)

Variable-Length Arrays

```
    puts( "\nOne-dimensional array:" );
    print1DArray( arraySize, array ); // pass 1-D VLA to function

    puts( "\nFirst two-dimensional array:" );
    print2DArray( row1, col1, array2D1 ); // pass 2-D VLA to function

    puts( "\nSecond two-dimensional array:" );
    print2DArray( row2, col2, array2D2 ); // pass other 2-D VLA to function
} // end main

void print1DArray( int size, int array[      ] )
{
    // output contents of array
    for ( int i = 0; i < size; i++ ) {
        printf( "array[%d] = %d\n", i, array[ i ] );
    } // end for
} // end function print1DArray

void print2DArray( int row, int col, int arr[     ][ col ] )
{
    // output contents of array
    for ( int i = 0; i < row; ++i ) {
        for ( int j = 0; j < col; ++j ) {
            printf( "%5d", arr[ i ][ j ] );
        } // end for

        puts( "" );
    } // end for
} // end function print2DArray
```

Fig. 6.23 | Using variable-length arrays in C99. (Part 4 of 5.)

Variable-Length Arrays

```
Enter size of a one-dimensional array: 6
Enter number of rows and columns in a 2-D array: 2 5
Enter number of rows and columns in another 2-D array: 4 3

sizeof(array) yields array size of 24 bytes

One-dimensional array:
array[0] = 0
array[1] = 1
array[2] = 4
array[3] = 9
array[4] = 16
array[5] = 25

First two-dimensional array:
    0    1    2    3    4
    1    2    3    4    5

Second two-dimensional array:
    0    1    2
    1    2    3
    2    3    4
    3    4    5
```

Fig. 6.23 | Using variable-length arrays in C99. (Part 5 of 5.)