VIET NAM NATIONAL UNIVERSITY HO CHI MINH CITY

HOCHIMINH CITY INTERNATIONAL UNIVERSITY • HCM - IU •

SCHOOL OF ELECTRICAL ENGINEERING * IU - EE *

Programming for Engineers

Lecture 5: Working with C Arrays

Course ID: EE057IU

Lecture Outline

6.1 Introduction

6.2 Arrays

6.3 Defining Arrays

6.4 Array Examples

6.5 Manipulate Strings

6.7 Passing Arrays to Functions

6.10 Searching Arrays

6.11 Multiple-Subscripted Arrays

Introduction

- **Arrays (Chapter 6)**
  - Part of data structure
  - Data structures of related data items – same types

- **Notion of *structs* (Chapter 10)**
  - Data structures of related data items – possibly different types

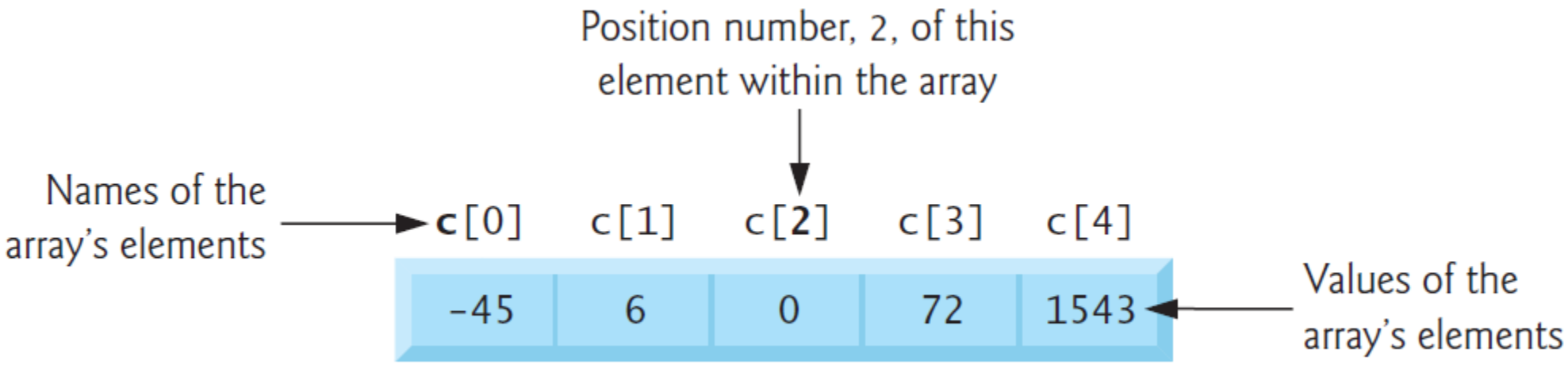
- Both **Arrays** and ***structs*** are "static" entities
  - remain same size thoughout their lifetimes

Arrays

- Data structures of related data items – **same types**
- A group of **consecutive memory locations** that all have the same type.
- To refer to an element, specify:
  - Array name
  - Position number in square brackets [ ]
- Format:

  arrayname[*position number*]
  - First element at position 0
  - n-element array named c: c[0], c[1] ... c[n–1]

Example Array



- Integer array called c
- First element is located at index 0

Array indexing

- First element in every array is zeroth element.
- Array name, like other identifiers, can contain only letters, digits, and underscore. Can not begin with a digit
- Position number inside bracket [ ] is called index or subscript.
- Index must be integer or integer expression
  - array_name[x], array_name[x+y], etc.

- For example, if a = 5 and b = 6, then the statement
  - c[a+b] +=2

  adds 2 to array element c[11]

Array in memory

- Array occupies contiguous space in memory
- The following definition reserves 5 elements for integer array c

```
int c[5];
```

- The definitions:

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

  - reserve 100 elements for integer array b and 27 elements for integer array x.
  - Array b index range 0 – 99
  - Array x index range 0 – 26

6.4.1 Initializing an array

```
1  // fig06_01.c
2  // Initializing the elements of an array to zeros.
3  #include <stdio.h>
4
5  // function main begins program execution
6  int main(void) {
7     int n[5]; // n is an array of five integers
8
9     // set elements of array n to 0
10    for (size_t i = 0; i < 5; ++i) {
11       n[i] = 0; // set element at location i to 0
12    }
13
14    printf("%s%8s\n", "Element", "Value");
15
16    // output contents of array n in tabular format
17    for (size_t i = 0; i < 5; ++i) {
18       printf("%7zu%8d\n", i, n[i]);
19    }
20 }
```

```
Element   Value
      0       0
      1       0
      2       0
      3       0
      4       0
```

- Counter-control variable size_t
- size_t is defined in <stddef.h>
- Recommended array's size or array's indices

- Conversion specification %zu is used to display size_t values.

6.4.2 Initializing with initializer list

```
1  // fig06_02.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     int n[5] = {32, 27, 64, 18, 95}; // initialize n with initializer list
8
9     printf("%s%8s\n", "Element", "Value");
10
11    // output contents of array in tabular format
12    for (size_t i = 0; i < 5; ++i) {
13       printf("%7zu%8d\n", i, n[i]);
14    }
15 }
```

```
Element   Value
      0      32
      1      27
      2      64
      3      18
      4      95
```

Fig. 6.2 | Initializing the elements of an array with an initializer list.

Initializing with fewer intializers

- If there are fewer initializers than elements in the array, the remaining elements are initialized to zero.
- Example:

```
// initializes entire array to zeros
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.

Initializing without array size

- 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.

6.4.3 Initializing to even list

```c
// fig06_03.c
// Initializing the elements of array s to the even integers from 2 to 10.
#include <stdio.h>
#define SIZE 5 // 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] = {0}; // array s has SIZE elements

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

   printf("%s%8s\n", "Element", "Value");

   // output contents of array s in tabular format
   for (size_t j = 0; j < SIZE; ++j) {
      printf("%7zu%8d\n", j, s[j]);
   }
}
```

```
Element   Value
      0       2
      1       4
      2       6
      3       8
      4      10
```

Preprocessor

- The #define preprocessor directive is introduced in this program.

  #define SIZE 5

  defines a symbolic constant SIZE whose value is 5.

- 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 modifiable.

6.4.4 Adding elements of an array

```c
// fig06_04.c
// Computing the sum of the elements of an array.
#include <stdio.h>
#define SIZE 5

// function main begins program execution
int main(void) {
   // use an initializer list to initialize the array
   int a[SIZE] = {1, 2, 3, 4, 5};
   int total = 0; // sum of array

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

   printf("The total of a's values is %d\n", total);
}
```

```
The total of a's values is 15
```

6.4.5 Using Arrays to Summarize Survey Results

- Consider the problem statement:

Twenty students (20) were asked to rate the quality of the food in the student cafeteria on a scale of 1 to 5 (1 means awful, and 5 means excellent). Place the 20 responses in an integer array and summarize the frequency of the rating points.

6.4.5 Using Arrays to Summarize Survey Results

```c
// fig06_05.c
// Analyzing a student poll.
#include <stdio.h>
#define RESPONSES_SIZE 20 // define array sizes
#define FREQUENCY_SIZE 6

// function main begins program execution
int main(void) {
   // place the survey responses in the responses array
   int responses[RESPONSES_SIZE] =
      {1, 2, 5, 4, 3, 5, 2, 1, 3, 1, 4, 3, 3, 3, 2, 3, 3, 2, 2, 5};

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

   // for each answer, select the value of an element of the array
   // responses and use that value as a subscript into the array
   // frequency to determine the element to increment
   for (size_t answer = 0; answer < RESPONSES_SIZE; ++answer) {
      ++frequency[responses[answer]];
   }

   // display results
   printf("%s%12s\n", "Rating", "Frequency");

   // output the frequencies in a tabular format
   for (size_t rating = 1; rating < FREQUENCY_SIZE; ++rating) {
      printf("%6zu%12d\n", rating, frequency[rating]);
   }
}
```

```
Rating    Frequency
     1            3
     2            5
     3            7
     4            2
     5            3
```

Fig. 6.5 | Analyzing a student poll.

6.4.6 Graphing Array element values with Bar Charts

```c
// fig06_06.c
// Displaying a bar chart.
#include <stdio.h>
#define SIZE 5

// function main begins program execution
int main(void) {
      // use initializer list to initialize array n
      int n[SIZE] = {19, 3, 15, 7, 11};

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

      // for each element of array n, output a bar of the bar chart
      for (size_t i = 0; i < SIZE; ++i) {
          printf("%7zu%13d%8s", i, n[i], "");

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

          puts(""); // end a bar with a newline
      }
}
```

```
Element        Value        Bar Chart
      0           19        *******************
      1            3        ***
      2           15        ***************
      3            7        *******
      4           11        ***********
```

Character Arrays & String Representation

- So far, the only string-processing capability we know is $printf$
- A string “hello” is really an array of individual characters in C

➢ For example,

```
char string1[] = “first”;
```

- Initializes elements of array *string1* to individual characters in the string literal “first”
- Size of array *string1* determined by the compiler
- String “first” contains 5 characters plus special *string-termination character* called null character ‘\0’. → total 6 characters
- All strings end with null character ‘\0’

Character Array Indexing

```
char string1[] = “first”;
```

- Previous string is equivalent to

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

- One can access a string’s individual characters directly using array index notion
- For example, string1[0] is the character ‘f’ and string1[3] is the character ‘s’

Scanning string

- One can input a string directly into a character array

```
char string2[20]; //contain 19 characters and one 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 "non-string" variables.
- The **&** is normally used to provide $scanf$ with a variable's location in memory so that a value can be stored there.

Scanning string

- 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 vulerability.
- 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*

Printing String

- 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.

Treating Character Arrays as String

```c
// fig06_08.c
// Treating character arrays as strings.
#include <stdio.h>
#define SIZE 20

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

   // prompt for string from user then read it into array string1
   printf("%s", "Enter a string (no longer than 19 characters): ");
   scanf("%19s", string1); // input no more than 19 characters

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

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

   puts("");
}
```

```
Enter a string (no longer than 19 characters): Hello there
string1 is: Hello
string2 is: string literal
string1 with spaces between characters is:
H e l l o
```

- String1 only print hello, It stops reading when it encounters a whitespace character

6.7 Passing Arrays to Functions

- To pass an array argument to a function, specify the array's name without any brackets.
- For example,

```
int hourlyTemperatures[HOURS_IN_A_DAY];
modifyArray(hourlyTemperatures, HOURS_IN_A_DAY);
```

- The array's name evaluates to the address of the first element of the array.
- The called function can modify the element values in the callers' original arrays.

Passing Arrays to Functions (1)

```c
// fig06_11.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

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

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

   puts(""); // outputs a newline

   modifyArray(a, SIZE); // pass array a to modifyArray by reference
   puts("The values of the modified array are:");

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

```

Passing Arrays to Functions (2)

```
32      // output value of a[3]
33      printf("\n\n\nEffects of passing array element "
34         "by value:\n\nThe value of a[3] is %d\n", a[3]);
35
36      modifyElement(a[3]); // pass array element a[3] by value
37
38      // output value of a[3]
39      printf("The value of a[3] is %d\n", a[3]);
40   }
41
42   // in function modifyArray, "b" points to the original array "a" in memory
43   void modifyArray(int b[], size_t size) {
44      // multiply each array element by 2
45      for (size_t j = 0; j < size; ++j) {
46         b[j] *= 2; // actually modifies original array
47      }
48   }
49
50   // in function modifyElement, "e" is a local copy of array element
51   // a[3] passed from main
52   void modifyElement(int e) {
53      e *= 2; // multiply parameter by 2
54      printf("Value in modifyElement is %d\n", e);
55   }
```

Passing Arrays to Functions (3)

```
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
```

Elements of the array are modified

Element of the array are NOT modified

Protecting Array Elements

- Function tryToModifyArray 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.

```
1  // in function tryToModifyArray, array b is const, so it cannot be
2  // used to modify its array argument in the caller
3  void tryToModifyArray(const int b[]) {
4     b[0] /= 2; // error
5     b[1] /= 2; // error
6     b[2] /= 2; // error
7  }
```

In-class practice

- Search an Array:

Write a program to initialize an array of size `SIZE=10` with an initializer list. Also get a value for `num1` from user.

Pass the array as well as `num1` to a function. Within the function, check each element of array whether it matches `num1`.

Replace the array's element with 1 (if match) or with 0 (if no match). Return and print the new array in main().

```
#define SIZE 10
int array[SIZE] = {5, 12, 23, 34, 45, 56, 67, 78, 89, 100};
```

Linear Search – Binary Search

- The linear searching method works well for *small* or *unsorted* arrays.
- However, for large arrays linear searching is *inefficient*.
- If the array is sorted, the high-speed binary search technique can be used.
- The binary search algorithm eliminates from consideration one-half of the elements in a sorted array after each comparison.

Binary Search – searching in a sorted array

- The algorithm locates the *middle* element of the array and compares it to the search key.
- If they're equal, the search key is found and the index of that element is returned.
- If they're not equal, the problem is reduced to searching *one-half* of the array.
- If the search key is less than the middle element of the array, the *first half* of the array is searched, otherwise the *second half* of the array is searched.
- https://www.cs.princeton.edu/courses/archive/fall06/cos226/demo/demo-bsearch.ppt

Binary – C code (1)

```c
// fig06_15.c
// Binary search of a sorted array.
#include <stdio.h>
#define SIZE 15

// function prototypes
int binarySearch(const int b[], int key, size_t low, size_t high);
void printHeader(void);
void printRow(const int b[], size_t low, size_t mid, size_t high);

// function main begins program execution
int main(void) {
   int a[SIZE] = {0}; // create array a

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

   printf("%s", "Enter a number between 0 and 28: ");
   int key = 0; // value to locate in array a
   scanf("%d", &key);

   printHeader();

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

   // display results
   if (result != -1) {
      printf("\n%d found at subscript %d\n", key, result);
   }
   else {
      printf("\n%d not found\n", key);
   }
}
```

```c

// function to perform binary search of an array
int binarySearch(const int b[], int key, size_t low, size_t high) {
   // loop until low subscript is greater than high subscript
   while (low <= high) {
      size_t middle = (low + high) / 2; // determine middle subscript

      // display subarray used in this loop iteration
      printRow(b, low, middle, high);

      // if key matches, return middle subscript
      if (key == b[middle]) {
         return middle;
      }
      else if (key < b[middle]) { // if key < b[middle], adjust high
         high = middle - 1; // next iteration searches low end of array
      }
      else { // key > b[middle], so adjust low
         low = middle + 1; // next iteration searches high end of array
      }
   } // end while

   return -1; // searchKey not found
}
```

Binary – C code (2)

```c
// Print a header for the output
void printHeader(void) {
   puts("\nSubscripts:");

   // output column head
   for (int i = 0; i < SIZE; ++i) {
      printf("%3d ", i);
   }

   puts(""); // start new line of output

   // output line of - characters
   for (int i = 1; i <= 4 * SIZE; ++i) {
      printf("%s", "-");
   }

   puts(""); // start new line of output
}

// Print one row of output showing the current
// part of the array being processed.
void printRow(const int b[], size_t low, size_t mid, size_t high) {
   // loop through entire array
   for (size_t i = 0; i < SIZE; ++i) {
      // display spaces if outside current subarray range
      if (i < low || i > high) {
         printf("%s", "    ");
      }
      else if (i == mid) { // display middle element
         printf("%3d*", b[i]); // mark middle value
      }
      else { // display other elements in subarray
         printf("%3d ", b[i]);
      }
   }

   puts(""); // start new line of output
}
```

Binary – C code (3)

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

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

25 not found
```

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

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

8 found at subscript 4
```

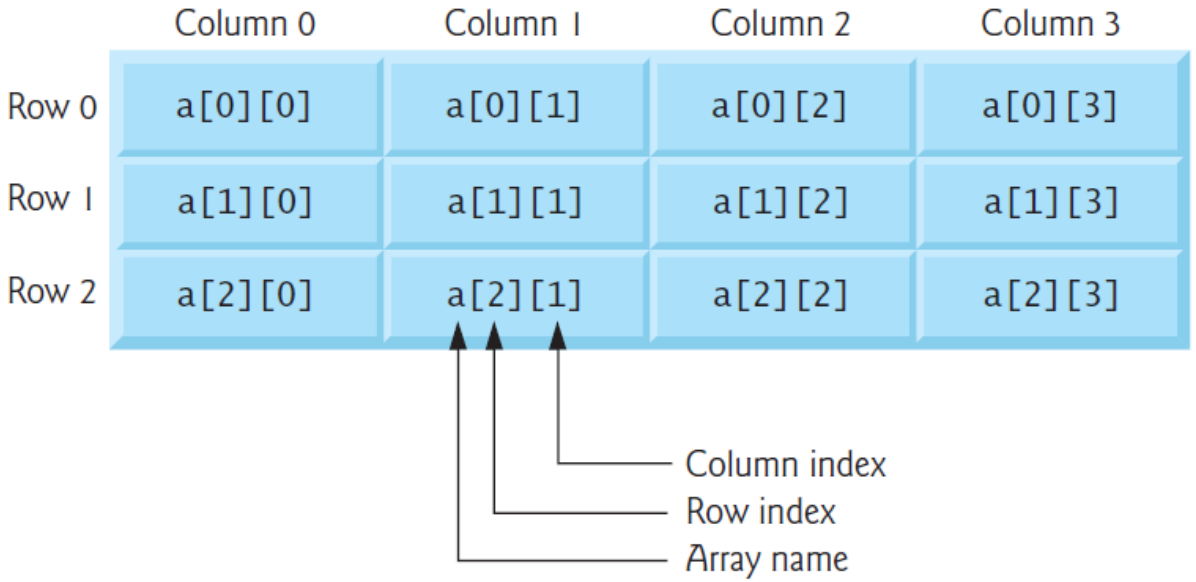
```
Enter a number between 0 and 28: 6

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

6 found at subscript 3
```

Multidimensional Arrays

- Arrays in C can have multiple indices.
- A common use of multidimensional arrays is to represent tables of values consisting of information arranged in rows and columns.
- Multidimensional arrays can have more than two indices.



Initialization

- Where it is defined
  - Braces for each dimension

```
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 initialized to 0.

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

  - 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.

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

Multidimensional Array example code

```c
// fig06_16.c
// Initializing multidimensional arrays.
#include <stdio.h>

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

// function main begins program execution
int main(void) {
   int array1[2][3] = {{1, 2, 3}, {4, 5, 6}};
   puts("Values in array1 by row are:");
   printArray(array1);

   int array2[2][3] = {{1, 2, 3}, {4, 5}};
   puts("Values in array2 by row are:");
   printArray(array2);

   int array3[2][3] = {{1, 2}, {4}};
   puts("Values in array3 by row are:");
   printArray(array3);
}
```

```c

// function to output array with two rows and three columns
void printArray(int a[][3]) {
   // loop through rows
   for (size_t i = 0; i <= 1; ++i) {
      // output column values
      for (size_t j = 0; j <= 2; ++j) {
         printf("%d ", a[i][j]);
      }

      printf("\n"); // start new line of output
   }
}
```

```
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
```

Two Dimensional Array Manipulation

- Example
  - studentGrades[3][4]
  - Row of the array represents a student.
  - Column represents a grade on one of the four exams the students took during the semester.
- The array manipulations are performed by four functions.
  - Function minimum determines the lowest grade of any student for the semester.
  - Function maximum determines the highest grade of any student for the semester.
  - Function average determines a particular student's semester average.
  - Function printArray outputs the two-dimensional array in a neat, tabular format.

Two Dimensional Array Manipulation

```c
// fig06_17.c
// Two-dimensional array manipulations.
#include <stdio.h>
#define STUDENTS 3
#define EXAMS 4

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

// function main begins program execution
int main(void) {
   // initialize student grades for three students (rows)
   int studentGrades[STUDENTS][EXAMS] =
      {{77, 68, 86, 73},
       {96, 87, 89, 78},
       {70, 90, 86, 81}};

   // output array studentGrades
   puts("The array is:");
   printArray(studentGrades, STUDENTS, EXAMS);

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

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

```c

// Find the minimum grade
int minimum(const int grades[][EXAMS], size_t pupils, size_t tests) {
   int lowGrade = 100; // initialize to highest possible grade

   // loop through rows of grades
   for (size_t row = 0; row < pupils; ++row) {
      // loop through columns of grades
      for (size_t column = 0; column < tests; ++column) {
         if (grades[row][column] < lowGrade) {
            lowGrade = grades[row][column];
         }
      }
   }

   return lowGrade; // return minimum grade
}

// Find the maximum grade
int maximum(const int grades[][EXAMS], size_t pupils, size_t tests) {
   int highGrade = 0; // initialize to lowest possible grade

   // loop through rows of grades
   for (size_t row = 0; row < pupils; ++row) {
      // loop through columns of grades
      for (size_t column = 0; column < tests; ++column) {
         if (grades[row][column] > highGrade) {
            highGrade = grades[row][column];
         }
      }
   }

   return highGrade; // return maximum grade
}
```

Two Dimensional Array Manipulation

```
71  // Determine the average grade for a particular student
72  double average(const int setOfGrades[], size_t tests) {
73     int total = 0; // sum of test grades
74
75     // total all grades for one student
76     for (size_t test = 0; test < tests; ++test) {
77        total += setOfGrades[test];
78     }
79
80     return (double) total / tests; // average
81  }
82
83  // Print the array
84  void printArray(const int grades[][EXAMS], size_t pupils, size_t tests) {
85     // output column heads
86     printf("%s", "                [0]  [1]  [2]  [3]");
87
88     // output grades in tabular format
89     for (size_t row = 0; row < pupils; ++row) {
90        // output label for row
91        printf("\nstudentGrades[%zu] ", row);
92
93        // output grades for one student
94        for (size_t column = 0; column < tests; ++column) {
95           printf("%-5d", grades[row][column]);
96        }
97     }
98  }
```

```
The array is:
                [0]  [1]  [2]  [3]
studentGrades[0] 77   68   86   73
studentGrades[1] 96   87   89   78
studentGrades[2] 70   90   86   81

Lowest grade: 68
Highest grade: 96
The average grade for student 0 is 76.00
The average grade for student 1 is 87.50
The average grade for student 2 is 81.75
```

In-class Practice

- Reverse an array

In-class Practice

- Represent a 2D array by a 1D array

In-class Practice

- Insert an element in a sorted array

In-class Practice

- Find second minimum