Programming for Engineers

Arrays

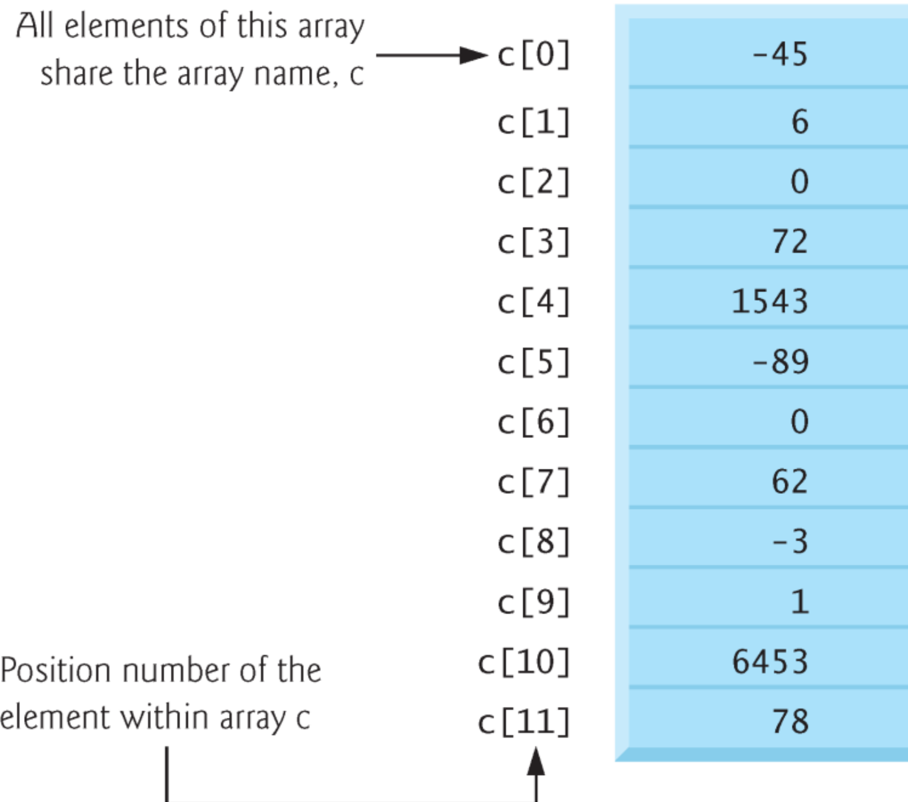
ICEN 200– Spring 2018

Prof. Dola Saha

Array

- Arrays are data structures consisting of related data items of the same type.
- A group of *contiguous* memory locations that all have the *same type*.
- To refer to a particular location or element in the array
  - Array’s name
  - Position number of the particular element in the array

Example Array



Array indexing

- The first element in every array is the zeroth element.
- An array name, like other identifiers, can contain only letters, digits and underscores and cannot begin with a digit.
- The position number within square brackets is called an index or subscript.
- An index must be an integer or an 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 12 elements for integer array c, which has indices in the range 0-11.
  - `int c[12];`
- The definition
  - `int b[100]; double x[27];`

  reserves 100 elements for integer array b and 27 elements for double array x.
- Like any other variables, uninitialized array elements contain garbage values.

Initializing array

```
1  // Fig. 6.3: fig06_03.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  {
8     int n[5]; // n is an array of five integers
9
10    // set elements of array n to 0
11    for (size_t i = 0; i < 5; ++i) {
12       n[i] = 0; // set element at location i to 0
13    }
14
15    printf("%s%13s\n", "Element", "Value");
16
17    // output contents of array n in tabular format
18    for (size_t i = 0; i < 5; ++i) {
19       printf("%7u%13d\n", i, n[i]);
20    }
21 }
```

Output

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

Use of `size_t`

- 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 recommended for any variable that represents an array's size or an array's indices.
- Type `size_t` is defined in header `<stddef.h>`, which is often included by other headers (such as `<stdio.h>`).
- [Note: If you attempt to compile Fig. 6.3 and receive errors, simply include `<stddef.h>` in your program.]

Initializing with initializer list

```
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[5] = {32, 27, 64, 18, 95};
10
11    printf("%s%13s\n", "Element", "Value");
12
13    // output contents of array in tabular format
14    for (size_t i = 0; i < 5; ++i) {
15       printf("%7u%13d\n", i, n[i]);
16    }
17 }
```

Output

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

Initializing with fewer initializers

- 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.
    ```

Initializing to even list

```
1   // Fig. 6.5: fig06_05.c
2   // Initializing the elements of array s to the even integers from 2 to 10.
3   #include <stdio.h>
4   #define SIZE 5 // maximum size of array
5
6   // function main begins program execution
7   int main(void)
8   {
9      // symbolic constant SIZE can be used to specify array size
10     int s[SIZE]; // array s has SIZE elements
11
12     for (size_t j = 0; j < SIZE; ++j) { // set the values
13        s[j] = 2 + 2 * j;
14     }
15
16     printf("%s%13s\n", "Element", "Value");
17
18     // output contents of array s in tabular format
19     for (size_t j = 0; j < SIZE; ++j) {
20        printf("%7u%13d\n", j, s[j]);
21     }
22  }
```

Output

```
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.

Common Programming Error 6.3

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

Adding elements of an array

```c
1  // Fig. 6.6: fig06_06.c
2  // Computing the sum of the elements of an array.
3  #include <stdio.h>
4  #define SIZE 12
5
6  // function main begins program execution
7  int main(void)
8  {
9     // use an initializer list to initialize the array
10    int a[SIZE] = {1, 3, 5, 4, 7, 2, 99, 16, 45, 67, 89, 45};
11    int total = 0; // sum of array
12
13    // sum contents of array a
14    for (size_t i = 0; i < SIZE; ++i) {
15       total += a[i];
16    }
17
18    printf("Total of array element values is %d\n", total);
19 }
```

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

Using Arrays to Summarize Poll (1)

```
1   // Fig. 6.7: fig06_07.c
2   // Analyzing a student poll.
3   #include <stdio.h>
4   #define RESPONSES_SIZE 40 // define array sizes
5   #define FREQUENCY_SIZE 11
6
7   // function main begins program execution
8   int main(void)
9   {
10     // initialize frequency counters to 0
11     int frequency[FREQUENCY_SIZE] = {0};
12
13     // place the survey responses in the responses array
14     int responses[RESPONSES_SIZE] = {1, 2, 6, 4, 8, 5, 9, 7, 8, 10,
15          1, 6, 3, 8, 6, 10, 3, 8, 2, 7, 6, 5, 7, 6, 8, 6, 7, 5, 6, 6,
16          5, 6, 7, 5, 6, 4, 8, 6, 8, 10};
17
18     // for each answer, select value of an element of array responses
19     // and use that value as an index in array frequency to
20     // determine element to increment
21     for (size_t answer = 0; answer < RESPONSES_SIZE; ++answer) {
22        ++frequency[responses[answer]];
23     }
24
```

Using Arrays to Summarize Poll (2)

```
25      // display results
26      printf("%s%17s\n", "Rating", "Frequency");
27
28      // output the frequencies in a tabular format
29      for (size_t rating = 1; rating < FREQUENCY_SIZE; ++rating) {
30         printf("%6d%17d\n", rating, frequency[rating]);
31      }
32   }
```

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

Histogram with Array elements (1)

```
1   // Fig. 6.8: fig06_08.c
2   // Displaying a histogram.
3   #include <stdio.h>
4   #define SIZE 5
5
6   // function main begins program execution
7   int main(void)
8   {
9      // use initializer list to initialize array n
10     int n[SIZE] = {19, 3, 15, 7, 11};
11
12     printf("%s%13s%17s\n", "Element", "Value", "Histogram");
13
14     // for each element of array n, output a bar of the histogram
15     for (size_t i = 0; i < SIZE; ++i) {
16        printf("%7u%13d        ", i, n[i]);
17
18        for (int j = 1; j <= n[i]; ++j) { // print one bar
19           printf("%c", '*');
20        }
21
22        puts(""); // end a histogram bar with a newline
23     }
24  }
```

Histogram with Array elements (1)

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

Character Arrays & String Representation

- Store *strings* in character arrays.
- So far, the only string-processing capability we have is outputting a string with `printf`.
- A string such as `"hello"` is really 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"`.

Size of Character Array

- In this case, the size of array `string1` is determined by the compiler based on the length of the string.
- The string `"first"` contains five characters *plus* a special *string-termination character* called the null character.
- Thus, array `string1` actually contains six elements.
- The character constant representing the null character is `'\0'`.
- All strings in C end with this character.

Character Array Indexing

- 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 index notation.
- For example, `string1[0]` is the character `'f'` and `string1[3]` is the character `'s'`.
    ```

Scanning string

- 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 & 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`.

Memory Management in Scanning String

- 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.

- Function `scanf` does *not* check how large the array is.

- Thus, `scanf` can write beyond the end of the array.

- You can use `gets(text)` to get the text from user.

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 (1)

```
1  // Fig. 6.10: fig06_10.c
2  // Treating character arrays as strings.
3  #include <stdio.h>
4  #define SIZE 20
5
6  // function main begins program execution
7  int main(void)
8  {
9     char string1[SIZE]; // reserves 20 characters
10    char string2[] = "string literal"; // reserves 15 characters
11
12    // read string from user into array string1
13    printf("%s", "Enter a string (no longer than 19 characters): ");
14    scanf("%19s", string1); // input no more than 19 characters
15
16    // output strings
17    printf("string1 is: %s\nstring2 is: %s\n"
18           "string1 with spaces between characters is:\n",
19           string1, string2);
```

Treating Character Arrays as String (2)

```
20
21      // output characters until null character is reached
22      for (size_t i = 0; i < SIZE && string1[i] != '\0'; ++i) {
23         printf("%c ", string1[i]);
24      }
25
26      puts("");
27   }
```

```
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
```

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 function call passes array `hourlyTemperatures` and its size to function `modifyArray`.
- The name of the array 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 Array to Functions (1)

```
1   // Fig. 6.13: fig06_13.c
2   // Passing arrays and individual array elements to functions.
3   #include <stdio.h>
4   #define SIZE 5
5
6   // function prototypes
7   void modifyArray(int b[], size_t size);
8   void modifyElement(int e);
9
10  // function main begins program execution
11  int main(void)
12  {
13     int a[SIZE] = {0, 1, 2, 3, 4}; // initialize array a
14
15     puts("Effects of passing entire array by reference:\n\nThe "
16        "values of the original array are:");
17
18     // output original array
19     for (size_t i = 0; i < SIZE; ++i) {
20        printf("%3d", a[i]);
21     }
22
23     puts(""); // outputs a newline
24
```

Passing Array to Functions (2)

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

Passing Array to Functions (3)

```
43  // in function modifyArray, "b" points to the original array "a"
44  // in memory
45  void modifyArray(int b[], size_t size)
46  {
47     // multiply each array element by 2
48     for (size_t j = 0; j < size; ++j) {
49        b[j] *= 2; // actually modifies original array
50     }
51  }
52
53  // in function modifyElement, "e" is a local copy of array element
54  // a[3] passed from main
55  void modifyElement(int e)
56  {
57     // multiply parameter by 2
58     printf("Value in modifyElement is %d\n", e *= 2);
59  }
```

Passing Array to Functions (4)

```
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
```

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   {
5      b[0] /= 2; // error
6      b[1] /= 2; // error
7      b[2] /= 2; // error
8   }
```

Classwork Assignment

- Search an Array: Write a program to initialize an array of size `S` 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. If it matches, return 1, else return 0 to the `main` function.

Binary Search – searching in a sorted array

- 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.

Demo

- [Demo from Princeton](https://www.cs.princeton.edu/courses/archive/fall06/cos226/demo/demo-bsearch.ppt)

https://www.cs.princeton.edu/courses/archive/fall06/cos226/demo/demo-bsearch.ppt

Binary Search – C code (1)

```
1  // Fig. 6.19: fig06_19.c
2  // Binary search of a sorted array.
3  #include <stdio.h>
4  #define SIZE 15
5
6  // function prototypes
7  size_t binarySearch(const int b[], int searchKey, size_t low, size_t high);
8  void printHeader(void);
9  void printRow(const int b[], size_t low, size_t mid, size_t high);
10
11 // function main begins program execution
12 int main(void)
13 {
14    int a[SIZE]; // create array a
15
16    // create data
17    for (size_t i = 0; i < SIZE; ++i) {
18       a[i] = 2 * i;
19    }
20
21    printf("%s", "Enter a number between 0 and 28: ");
22    int key; // value to locate in array a
23    scanf("%d", &key);
24
```

Binary Search – C code (2)

```
25      printHeader();
26
27      // search for key in array a
28      size_t result = binarySearch(a, key, 0, SIZE - 1);
29
30      // display results
31      if (result != -1) {
32         printf("\n%d found at index %d\n", key, result);
33      }
34      else {
35         printf("\n%d not found\n", key);
36      }
37   }
38
39   // function to perform binary search of an array
40   size_t binarySearch(const int b[], int searchKey, size_t low, size_t high)
41   {
42      // loop until low index is greater than high index
43      while (low <= high) {
44
45         // determine middle element of subarray being searched
46         size_t middle = (low + high) / 2;
47
```

Binary Search – C code (3)

```
48          // display subarray used in this loop iteration
49          printRow(b, low, middle, high);
50
51          // if searchKey matched middle element, return middle
52          if (searchKey == b[middle]) {
53             return middle;
54          }
55
56          // if searchKey is less than middle element, set new high
57          else if (searchKey < b[middle]) {
58             high = middle - 1; // search low end of array
59          }  if
60
61          // if searchKey is greater than middle element, set new low
62          else {
63             low = middle + 1; // search high end of array
64          }
65       } // end while
66
67       return -1; // searchKey not found
68    }
69
```

Binary Search – C code (4)

```
70   // Print a header for the output
71   void printHeader(void)
72   {
73      puts("\nIndices:");
74
75      // output column head
76      for (unsigned int i = 0; i < SIZE; ++i) {
77         printf("%3u ", i);
78      }
79
80      puts(""); // start new line of output
81
82      // output line of - characters
83      for (unsigned int i = 1; i <= 4 * SIZE; ++i) {
84         printf("%s", "-");
85      }
86
87      puts(""); // start new line of output
88   }
89
```

Binary Search – C code (5)

```
90  // Print one row of output showing the current
91  // part of the array being processed.
92  void printRow(const int b[], size_t low, size_t mid, size_t high)
93  {
94     // loop through entire array
95     for (size_t i = 0; i < SIZE; ++i) {
96
97        // display spaces if outside current subarray range
98        if (i < low || i > high) {
99           printf("%s", "    ");
100       }
101       else if (i == mid) { // display middle element
102          printf("%3d*", b[i]); // mark middle value
103       }
104       else { // display other elements in subarray
105          printf("%3d ", b[i]);
106       }
107    }
108
109    puts(""); // start new line of output
110 }
```

Binary Search – C code (6)

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

Indices:
  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
```

Binary Search – C code (7)

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

Indices:
  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 index 4
```

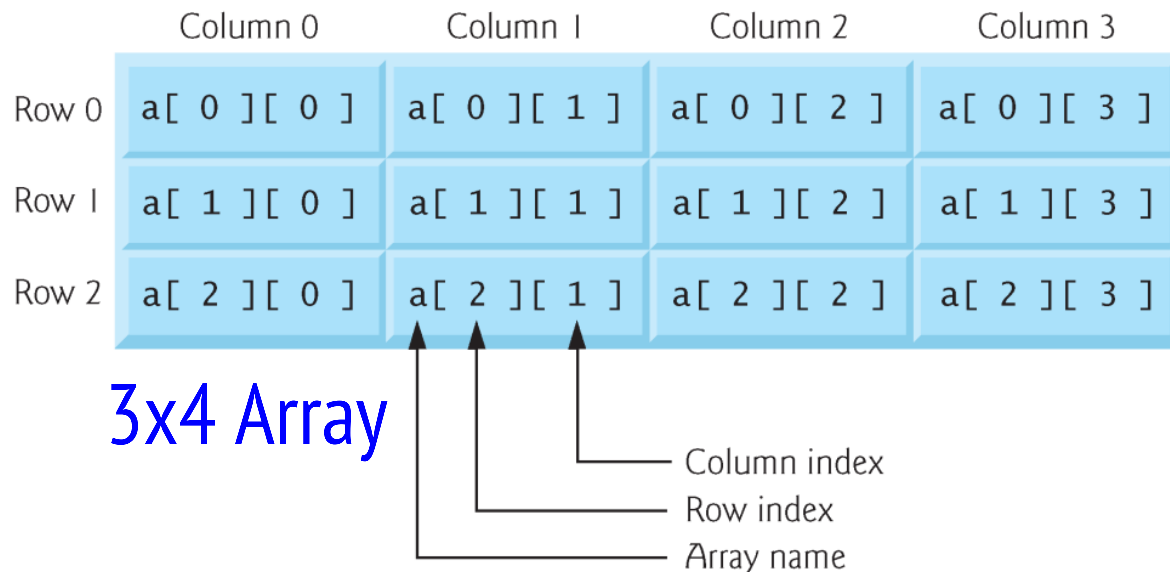
```
Enter a number between 0 and 28: 6

Indices:
  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 index 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 (1)

```
1   // Fig. 6.21: fig06_21.c
2   // Initializing multidimensional arrays.
3   #include <stdio.h>
4
5   void printArray(int a[][3]); // function prototype
6
7   // function main begins program execution
8   int main(void)
9   {
10      int array1[2][3] = {{1, 2, 3}, {4, 5, 6}};
11      puts("Values in array1 by row are:");
12      printArray(array1);
13
14      int array2[2][3] = {1, 2, 3, 4, 5};
15      puts("Values in array2 by row are:");
16      printArray(array2);
17
18      int array3[2][3] = {{1, 2}, {4}};
19      puts("Values in array3 by row are:");
20      printArray(array3);
21  }
22
```

Multidimensional Array Example Code (2)

```
23  // function to output array with two rows and three columns
24  void printArray(int a[][3])
25  {
26     // loop through rows
27     for (size_t i = 0; i <= 1; ++i) {
28
29        // output column values
30        for (size_t j = 0; j <= 2; ++j) {
31           printf("%d ", a[i][j]);
32        }
33
34        printf("\n"); // start new line of output
35     }
36  }
```

```
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.

2D Array Manipulation Code (1)

```
1   // Fig. 6.22: fig06_22.c
2   // Two-dimensional array manipulations.
3   #include <stdio.h>
4   #define STUDENTS 3
5   #define EXAMS 4
6
7   // function prototypes
8   int minimum(const int grades[][EXAMS], size_t pupils, size_t tests);
9   int maximum(const int grades[][EXAMS], size_t pupils, size_t tests);
10  double average(const int setOfGrades[], size_t tests);
11  void printArray(const int grades[][EXAMS], size_t pupils, size_t tests);
12
13  // function main begins program execution
14  int main(void)
15  {
16     // initialize student grades for three students (rows)
17     int studentGrades[STUDENTS][EXAMS] =
18        { { 77, 68, 86, 73 },
19          { 96, 87, 89, 78 },
20          { 70, 90, 86, 81 } };
21
22     // output array studentGrades
23     puts("The array is:");
24     printArray(studentGrades, STUDENTS, EXAMS);
```

2D Array Manipulation Code (2)

```
25
26      // determine smallest and largest grade values
27      printf("\n\nLowest grade: %d\nHighest grade: %d\n",
28         minimum(studentGrades, STUDENTS, EXAMS),
29         maximum(studentGrades, STUDENTS, EXAMS));
30
31      // calculate average grade for each student
32      for (size_t student = 0; student < STUDENTS; ++student) {
33         printf("The average grade for student %u is %.2f\n",
34            student, average(studentGrades[student], EXAMS));
35      }
36   }
37
```

2D Array Manipulation Code (3)

```
38  // Find the minimum grade
39  int minimum(const int grades[][EXAMS], size_t pupils, size_t tests)
40  {
41     int lowGrade = 100; // initialize to highest possible grade
42
43     // loop through rows of grades
44     for (size_t i = 0; i < pupils; ++i) {
45
46        // loop through columns of grades
47        for (size_t j = 0; j < tests; ++j) {
48
49           if (grades[i][j] < lowGrade) {
50              lowGrade = grades[i][j];
51           }
52        }
53     }
54
55     return lowGrade; // return minimum grade
56  }
57
```

2D Array Manipulation Code (4)

```
58  // Find the maximum grade
59  int maximum(const int grades[][EXAMS], size_t pupils, size_t tests)
60  {
61     int highGrade = 0; // initialize to lowest possible grade
62
63     // loop through rows of grades
64     for (size_t i = 0; i < pupils; ++i) {
65
66        // loop through columns of grades
67        for (size_t j = 0; j < tests; ++j) {
68
69           if (grades[i][j] > highGrade) {
70              highGrade = grades[i][j];
71           }
72        }
73     }
74
75     return highGrade; // return maximum grade
76  }
77
```

2D Array Manipulation Code (5)

```
78  // Determine the average grade for a particular student
79  double average(const int setOfGrades[], size_t tests)
80  {
81     int total = 0; // sum of test grades
82
83     // total all grades for one student
84     for (size_t i = 0; i < tests; ++i) {
85        total += setOfGrades[i];
86     }
87
88     return (double) total / tests; // average
89  }
90
```

2D Array Manipulation Code (6)

```
91   // Print the array
92   void printArray(const int grades[][EXAMS], size_t pupils, size_t tests)
93   {
94      // output column heads
95      printf("%s", "                 [0]  [1]  [2]  [3]");
96
97      // output grades in tabular format
98      for (size_t i = 0; i < pupils; ++i) {
99
100        // output label for row
101        printf("\nstudentGrades[%u] ", i);
102
103        // output grades for one student
104        for (size_t j = 0; j < tests; ++j) {
105           printf("%-5d", grades[i][j]);
106        }
107     }
108  }
```

2D Array Manipulation Code (7)

```
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
```

Lab Assignment

- Matrix Addition/Subtraction – two matrices should have same number of rows and columns.

$$\mathbf{A}+\mathbf{B} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mn} \end{bmatrix}$$
$$= \begin{bmatrix} a_{11}+b_{11} & a_{12}+b_{12} & \cdots & a_{1n}+b_{1n} \\ a_{21}+b_{21} & a_{22}+b_{22} & \cdots & a_{2n}+b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}+b_{m1} & a_{m2}+b_{m2} & \cdots & a_{mn}+b_{mn} \end{bmatrix}$$

Addition

$$\begin{bmatrix} 1 & 3 \\ 1 & 0 \\ 1 & 2 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 7 & 5 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 1+0 & 3+0 \\ 1+7 & 0+5 \\ 1+2 & 2+1 \end{bmatrix} = \begin{bmatrix} 1 & 3 \\ 8 & 5 \\ 3 & 3 \end{bmatrix}$$

Subtraction

$$\begin{bmatrix} 1 & 3 \\ 1 & 0 \\ 1 & 2 \end{bmatrix} - \begin{bmatrix} 0 & 0 \\ 7 & 5 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 1-0 & 3-0 \\ 1-7 & 0-5 \\ 1-2 & 2-1 \end{bmatrix} = \begin{bmatrix} 1 & 3 \\ -6 & -5 \\ -1 & 1 \end{bmatrix}$$

https://en.wikipedia.org/wiki/Matrix_addition

Matrix Multiplication

- If A is a $n \times m$ matrix and B is a $m \times p$ matrix, then Matrix Multiplication is given by following formula

$$\mathbf{A} = \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nm} \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} B_{11} & B_{12} & \cdots & B_{1p} \\ B_{21} & B_{22} & \cdots & B_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ B_{m1} & B_{m2} & \cdots & B_{mp} \end{pmatrix}$$

$$\mathbf{AB} = \begin{pmatrix} (\mathbf{AB})_{11} & (\mathbf{AB})_{12} & \cdots & (\mathbf{AB})_{1p} \\ (\mathbf{AB})_{21} & (\mathbf{AB})_{22} & \cdots & (\mathbf{AB})_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ (\mathbf{AB})_{n1} & (\mathbf{AB})_{n2} & \cdots & (\mathbf{AB})_{np} \end{pmatrix}$$

$$(\mathbf{AB})_{ij} = \sum_{k=1}^{m} A_{ik} B_{kj}$$

https://en.wikipedia.org/wiki/Matrix_multiplication

Matrix Multiplication - Illustrated

$$\mathbf{AB} = \begin{pmatrix} a & b & c \\ x & y & z \end{pmatrix} \begin{pmatrix} \alpha & \rho \\ \beta & \sigma \\ \gamma & \tau \end{pmatrix} = \begin{pmatrix} a\alpha + b\beta + c\gamma & a\rho + b\sigma + c\tau \\ x\alpha + y\beta + z\gamma & x\rho + y\sigma + z\tau \end{pmatrix}$$

$$\mathbf{BA} = \begin{pmatrix} \alpha & \rho \\ \beta & \sigma \\ \gamma & \tau \end{pmatrix} \begin{pmatrix} a & b & c \\ x & y & z \end{pmatrix} = \begin{pmatrix} \alpha a + \rho x & \alpha b + \rho y & \alpha c + \rho z \\ \beta a + \sigma x & \beta b + \sigma y & \beta c + \sigma z \\ \gamma a + \tau x & \gamma b + \tau y & \gamma c + \tau z \end{pmatrix}$$

Variable Length Array

- In early versions of C, all arrays had constant size.
- If size is unknown at compilation time
  - Use dynamic memory allocation with `malloc`

- The C standard allows a variable-length array
  - An array whose length, or size, is defined in terms of an expression evaluated at execution time.

Variable Length Array Code (1)

```
1  // Fig. 6.23: fig06_23.c
2  // Using variable-length arrays in C99
3  #include <stdio.h>
4
5  // function prototypes
6  void print1DArray(size_t size, int array[size]);
7  void print2DArray(int row, int col, int array[row][col]);
8
9  int main(void)
10 {
11    printf("%s", "Enter size of a one-dimensional array: ");
12    int arraySize; // size of 1-D array
13    scanf("%d", &arraySize);
14
15    int array[arraySize]; // declare 1-D variable-length array
16
17    printf("%s", "Enter number of rows and columns in a 2-D array: ");
18    int row1, col1; // number of rows and columns in a 2-D array
19    scanf("%d %d", &row1, &col1);
20
21    int array2D1[row1][col1]; // declare 2-D variable-length array
22
```

Variable Length Array Code (2)

```
23       printf("%s",
24          "Enter number of rows and columns in another 2-D array: ");
25       int row2, col2; // number of rows and columns in another 2-D array
26       scanf("%d %d", &row2, &col2);
27
28       int array2D2[row2][col2]; // declare 2-D variable-length array
29
30       // test sizeof operator on VLA
31       printf("\nsizeof(array) yields array size of %d bytes\n",
32          sizeof(array));
33
34       // assign elements of 1-D VLA
35       for (size_t i = 0; i < arraySize; ++i) {
36          array[i] = i * i;
37       }
38
39       // assign elements of first 2-D VLA
40       for (size_t i = 0; i < row1; ++i) {
41          for (size_t j = 0; j < col1; ++j) {
42             array2D1[i][j] = i + j;
43          }
44       }
45
```

Variable Length Array Code (3)

```
46      // assign elements of second 2-D VLA
47      for (size_t i = 0; i < row2; ++i) {
48         for (size_t j = 0; j < col2; ++j) {
49            array2D2[i][j] = i + j;
50         }
51      }
52
53      puts("\nOne-dimensional array:");
54      print1DArray(arraySize, array); // pass 1-D VLA to function
55
56      puts("\nFirst two-dimensional array:");
57      print2DArray(row1, col1, array2D1); // pass 2-D VLA to function
58
59      puts("\nSecond two-dimensional array:");
60      print2DArray(row2, col2, array2D2); // pass other 2-D VLA to function
61   }
62
63   void print1DArray(size_t size, int array[size])
64   {
65      // output contents of array
66      for (size_t i = 0; i < size; i++) {
67         printf("array[%d] = %d\n", i, array[i]);
68      }
69   }
```

Variable Length Array Code (4)

```
70
71  void print2DArray(size_t row, size_t col, int array[row][col])
72  {
73     // output contents of array
74     for (size_t i = 0; i < row; ++i) {
75        for (size_t j = 0; j < col; ++j) {
76           printf("%5d", array[i][j]);
77        }
78
79        puts("");
80     }
81  }
```

Variable Length Array Code (5)

```
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
```

Scan string with space

- Function `scanf` will read characters until a *space, tab, newline* or *end-of-file indicator* is encountered.
- Use `fgets` function.
  - `char *fgets(char *str, int n, FILE *stream)`
  - `str` – character array
  - `n` – maximum number of characters to be read
  - `stream` – where we are reading the data from

```
char buf[100];
fgets(buf, 100, stdin);
printf("string is: %s\n", buf);
```

Classwork

- Reverse an array

Classwork

- Represent a 2D array by a 1D array

Classwork

- Insert an element in a sorted array

Classwork

- Find second minimum