Multi-dimensional Arrays
3/16/01   Lecture #16    16.070

• Review:
  ➢ An array is set of elements that all have same data type
  ➢ Array elements stored sequentially in memory and accessed using integer index
  ➢ First element has index of 0

• Arrays can be of multiple dimensions: 1-D, 2-D, 3-D, etc.
  ➢ Arrays can be declared in which each element is itself an array
Visualizing Multi-dimensional Arrays

• Draw a One-Dimensional Array of 8 elements

Visualizing Multi-dimensional Arrays - cont.

• Draw a Two-Dimensional Array of 8 elements, each containing 5 elements
Visualizing Multi-dimensional Arrays - cont.

- Draw a Three-Dimensional Array of 8 elements, each contain 5 elements, and each of those contain 3 elements
Visualizing Multi-dimensional Arrays

- Draw a Four-Dimensional Array of 8 elements, each contain 5 elements, each of those contain 3 elements, each of those contain 2 elements
Declaring Multi-Dimensional Arrays

- Multi-dimensional arrays must be declared, just like variables and one-dimensional arrays

- Each dimension is represented by a subscript: [ ], [ ][ ], [ ][ ][ ], etc.

- For a 2-D array, first subscript defines the Row Number, second subscript defines the Column Number

- Format for declaring a 2-D array

  `<type> <array_name> [<#of_rows>][<#of_columns>]`;

- Example declaration

  ```
  int grades [students] [exams];
  ```
Multiple Dimensional Arrays - Example

- Create a 2-D array to represent the torque of 4 reaction wheels. Each wheel has a force component in each s/c axis (roll, pitch, yaw)

```c
float wheels [4][3];  /* 4 wheels x 3 axes */
```

Index for each element in `wheels` array

<table>
<thead>
<tr>
<th>Row\Col</th>
<th>1: Roll</th>
<th>2: Pitch</th>
<th>3: Yaw</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: RW1</td>
<td>[0][0]</td>
<td>[0][1]</td>
<td>[0][2]</td>
</tr>
<tr>
<td>2: RW2</td>
<td>[1][0]</td>
<td>[1][1]</td>
<td>[1][2]</td>
</tr>
<tr>
<td>3: RW3</td>
<td>[2][0]</td>
<td>[2][1]</td>
<td>[2][2]</td>
</tr>
<tr>
<td>4: RW4</td>
<td>[3][0]</td>
<td>[3][1]</td>
<td>[3][2]</td>
</tr>
</tbody>
</table>

- Two dimensional array is a convenient way of visualizing the data
- However, internally the data are stored sequentially, by rows.

```
[0][0]  [0][1]  [0][2]  [1][0]  [1][1]  [1][2] ...
```
Initializing Multiple Dimensional Arrays

- Like variables and single-dimensional arrays, multi-dimensional arrays can be initialized at compile time or at run time
  - At compile time, enclose each row in braces, and enclose all rows by one outer set of braces (for 2-D arrays)

    ```
    float wheels[4][3] = {
      {0.0, 0.0, 0.0},
      {0.0, 0.0, 0.0},
      {0.0, 0.0, 0.0},
      {0.0, 0.0, 0.0}
    }
    ```

  - At run time, loop over each index: use nested for loops

    ```
    for (i = 0; i < 3; i++)
      for (j = 0; j < 2; j++)
        wheels[i][j] = 0.0;
    ```

  - Generalize to higher dimensions: to initialize values in an N-dimensional array at run time, iterate over each index, usually right-most index first

- Like variables, un-initialized arrays contain garbage!
Manipulating Multi-Dimensional Arrays - Examples

• Declare a 3x2 array

```c
const int rows = 3;
const int cols = 2;
int matrix [rows][cols] = {
    {5, 7},
    {2, 8},
    {10, 4},
};
```

➢ Sum up rows:

```c
for (i = 0; i < rows; i++)
{
    sum = 0;
    for (j = 0; j < cols; j++)
        sum = sum + matrix[i][j];
    printf ("Sum for row %d is %d\n", i, sum);
}
```

➢ Sum up columns:
Manipulating Multi-Dimensional Arrays - Example

- Calculate total rainfall for each of 5 years, based on monthly averages

```c
#include <stdio.h>
#define MONTHS 12 /* number of months in year */
#define YRS 3     /* number of years of data */
int main(void) {
    /* initialize rainfall data for 1998-2000 */
    float rain[YRS][MONTHS] = {
    {10.2, 8.1, 6.8, 4.2, 2.1, 1.8, 0.2, 0.3, 1.1, 2.1, 6.1, 7.4},
    {9.2, 9.8, 4.4, 3.3, 2.2, 0.8, 0.4, 0.0, 0.1, 1.2, 2.5, 5.3},
    {8.6, 5.6, 1.3, 1.5, 2.5, 2.0, 0.5, 0.4, 0.9, 0.3, 2.1, 3.5}
    };
    int year = 0, month = 0;
    float subtot = 0.0, total = 0.0;
    printf (" YEAR    RAINFALL   (inches)\n");
    for (year = 0; year < YRS; year++) {
        /* for each year, sum rainfall over all months */
        for (month = 0; month < MONTHS; month++)
            subtot = subtot + rain[year][month];
        printf ("%d   %f\n", 1998 + year, subtot);
        total = total + subtot; /* total for all years */
    }
    printf ("\nTotal rainfall for all years was %f inches.\n", total);
    return 0;
}
```
Passing 1-D Arrays to Functions

- Name of array is the address of the **first element** in array
  
  ➢ For one-D arrays, name of array points to an element which is the zero index entry of the array

```c
const int axes = 3;
float sc_torque [axes] = {0.0, 0.1, 0.2};

sc_torque \rightarrow \begin{array}{c}
0.0 \\
1.0 \\
2.0
\end{array}
```

➢ In calling statement, name of array is passed without subscript

```c
total_torque = calc_torque (sc_torque, axes)
```

➢ In function definition, declare formal argument as a pointer to initial element of array

```c
float calc_torque (float torques[], int num_axes)
```
Passing 2-D Arrays to Functions

• For two-D arrays, name of array points to the zero index entry, which is the first row of the 2-D array

```c
float wheels [4][3] =
{
    {0.0, 0.1, 0.2},
    {1.0, 1.1, 1.2},
    {2.0, 2.1, 2.2},
    {3.0, 3.1, 3.2}
};
```

wheels →

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0, 0.1, 0.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.1, 1.1, 1.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.2, 2.1, 2.2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Passing 2-D Arrays to Functions - cont.

- In calling statement, name of array is passed without subscript
  
  \[ \text{rates} = \text{calc\_rates} \left( \text{wheels, 4} \right) \]

- In function definition, must declare second subscript of formal parameter
  
  \[ \text{float calc\_rates} \left( \text{float wheels[][][3], int num\_wheels} \right); \]
  
  /*prototype*/

  - Compiler needs to know size of each element (i.e., size of each row for a 2-D)
  - You may omit size of array being passed, but must specify size of each element
  - In general, may omit only the first size specification, but must specify other sizes
Passing 2-D Arrays to Functions - Example

• Examine the following example

```c
float two_axes_gyro_bias[3][2] = {
    {0.01, 0.02},
    {0.03, 0.02},
    {0.01, 0.03},
};
/*xy, yz, xz gyro biases*/
two_axes_gyro_bias == ? address of array of 2 floats = &t_a_g_b[0]
two_axes_gyro_bias[0] == ? address of a float = &t_a_g_b[0][0]

➢ Same value?

two_axes_gyro_bias + 1 == ? refers to 2 float object
two_axes_gyro_bias[0] + 1 == ? refers to a float

➢ Same value?

two_axes_gyro_bias[0][0] == ? 0.01
```
Protecting Array Contents

- When passing information to a function, pass by value or pass by reference (pointer)
  - Pass by value preserves contents of original variable since value is copied into a local variable
  - Pass by pointer allows function to have access to original variable. Integrity of constant may be compromised
  - Arrays are passed to functions by pointer (more efficient)
  - Array can be declared constant inside function to prevent function from modifying contents, even if array is not declared constant outside of function

```c
float total_torque (const float wheels[][num_axes],
                  int num_wheels);
```

- If program attempts to modify contents of constant array, compiler will identify error
Constant Arrays

- Like variables, arrays can be declared as constant
  - Constant arrays are a good way to represent look-up tables
    ```
    const float wheels[4][3] = {
        {0.90, 0.05, 0.02},
        {0.03, 0.90, 0.01},
        {0.02, 0.02, 0.90},
        {0.34, 0.32, 0.32}
    };
    ```
  - Compiler will guard against the value of a constant array being changed
    - Attempts to alter array contents will generate syntax error
    - Compiler probably will not guard against mis-handled pointers
    - Compiler probably will not prevent another array, whose limits are incorrectly defined, from overwriting a neighboring constant (array or otherwise)
Review

• Multi-dimensional arrays are useful for storing/manipulating multi-vectored data of the same type (e.g., monthly rainfall over n years)

• Have care when iterating over subscripts -- order is important!

• Read Sections C11.8-11.10 to solidify these concepts

• Extra help session offered Sunday, 3/18. Consider starting homework prior to this session and bring questions

• Incentive proposal for exams: going once, going twice…?

• I have Exam #1 exams that have not been picked up yet. Come see me after class