CS1100
Introduction to Programming

Multi-Dimensional Arrays

Madhu Mutyam
Department of Computer Science and Engineering
Indian Institute of Technology Madras

Multi-dimensional Arrays

- char multi[4][10];
  - multi[4] – an array of ten characters
  - 4 such arrays
    - multi[0] = {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9'}
    - multi[1] = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j'}
    - multi[2] = {'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J'}
    - multi[3] = {'9', '8', '7', '6', '5', '4', '3', '2', '1', '0'}
- Individual elements are addressable:
  - multi[0][3] = '3'

Ted Jenson's tutorial on pointers
http://pweb.netcom.com/~tjensen/ptr/cpoint.htm

Linear Contiguous Memory

multi[0] = {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9'}
multi[1] = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j'}
multi[2] = {'A','B','C', 'D', 'E', 'F', 'G', 'H', 'I', 'J'}
multi[3] = {'9', '8', '7', '6', '5', '4', '3', '2', '1', '0'}

- The data is stored in the memory as

0123456789abcdefghijABCDEFGHIJ9876543210

↑

starting at the address &multi[0][0]

Address Pointers

multi[0] = {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9'}
multi[1] = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j'}
multi[2] = {'A','B','C', 'D', 'E', 'F', 'G', 'H', 'I', 'J'}
multi[3] = {'9', '8', '7', '6', '5', '4', '3', '2', '1', '0'}

- *multi* is the address of location with '0'
- *multi*+1 is the address of 'a'
  - It adds 10, the number of columns, to get this location. If we were dealing with integers and an array with the same dimension the compiler would add 10***sizeof*(*int*)**
- To get to the content of the 2nd element in the 4th row we need to use *(*(*multi* + 3) + 1)

Address Computation

- With a little thought we can see that:

  ((*multi* + row) + col) and

  multi[row][col]

  yield the same results
- Because of the double de-referencing required in the pointer version, the name of a 2-dimensional array is often said to be equivalent to a pointer to a pointer

multi[*row*][*col*] ≡ *(*(*multi* + *row*) + *col*)

- To understand more fully what is going on, let us replace

  *(*multi* + *row*) with X

  i.e., *(*(*multi*+*row*)+*col*) = *(X + *col*)
- X is like a pointer since the expression is de-referenced and *col* is an integer
- Here "pointer arithmetic" is used
- That is, the address pointed to (i.e., value of)

  X + *col* = X + *col* * ***sizeof*(*int*)**

multi[*row*][*col*] ≡ *(*(*multi* + *row*) + *col*)

- Since we know the memory layout for 2 dimensional arrays, we can determine that in the expression (*multi* + *row*) as used above, (*multi* + *row* + 1) must increase by an amount equal to that needed to "point to" the next row, which for integers would be an amount equal to
  COLS * sizeof(int)
- That says that if the expression *(*(*multi* + *row*) + *col*) is to be evaluated correctly at run time, the compiler must generate code which takes into consideration the value of COLS, i.e., the 2nd dimension

remember passing arrays as parameters?

multi[*row*][*col*] ≡ *(*(*multi* + *row*) + *col*)

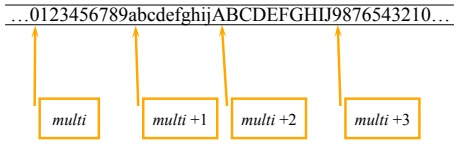
- Thus, to evaluate either expression, a total of 5 values must be known:
  - Address of the first element of the array, which is returned by the expression *multi*, i.e., name of the array
  - The size of the type of the elements of the array, in this case, ***sizeof*(*int*)**
  - The 2nd dimension of the array
  - The specific index value for the first dimension, *row* in this case
  - The specific index value for the second dimension, *col* in this case

Ted Jenson's tutorial on pointers
http://pweb.netcom.com/~tjensen/ptr/cpoint.htm

multi[*row*][*col*] ≡ *(*(*multi* + *row*) + *col*)

- Question:
  - When we say value = **ptr*; the pointer *ptr* is de-referenced to get the data stored
  - What happens in *(*(*array* + *row*) + *column*)?
  - Why is *(*array* + *row*) not de-referenced to give, say, an integer?
- Answer:
  - It is de-referenced
  - Remember a 2-D array is a pointer to a pointer
  - *(*array* + *row*) de-references to a pointer to a 1-D *array*
  - *(*array* + *row*) + 1 would do a pointer increment

Array *multi*[4][10]



Address(*multi*+1) = address(*multi*) + 10
Because the character array has 10 columns
Each of these is a pointer to a pointer (or pointer to a 1-D array)

Revisiting Functions on Arrays

- Initializing a 2-dimensional array

```
void set_value(int m_array[][COLS])
{
        int row, col;
        for (row = 0; row < ROWS; row++)
          {
            for (col = 0; col < COLS; col++)
                {
                  m_array[row][col] = 1;
                }
          }
}
```


Recap

- Arrays
- Functions
- Sorting
- Pointers
- Strings

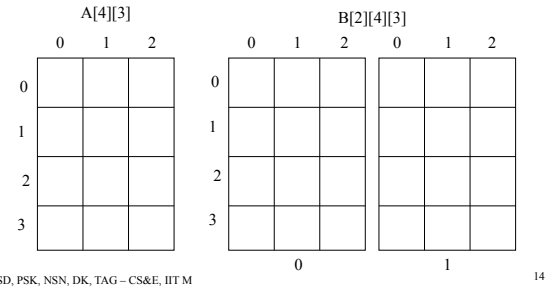
Arrays

- A data structure containing items of same data type
- Declaration: array name, storage reservation
  - ***int*** marks[7] = {22,15,75,56,10,33,45};
    - a contiguous group of memory locations
    - named "marks" for holding 7 integer items
  - elements/components - variables
    - marks[0], marks[1], … , marks[6]
    - marks[*i*], where *i* is a position/subscript (0≤i≤6)
  - the value of marks[2] is 75
  - new values can be assigned to elements
    - marks[3] = 36;

22	0
15	1
75	2
56	3
10	4
33	5
45	6

Multi-Dimensional Arrays

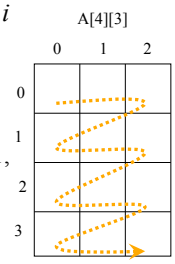
- Arrays with two or more dimensions can be defined



Two Dimensional Arrays

- Declaration: int A[4][3] : 4 rows and 3 columns, 4×3 array
- Elements: A[*i*][*j*] - element in row *i* and column *j* of array A
- Rows/columns numbered from 0
- Storage: row-major ordering
  - elements of row 0, elements of row 1, etc
- Initialization:

  int B[2][3]={{4,5,6},{0,3,5}};



Functions

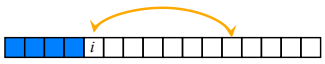
- Break large computing tasks into small ones
- Transfer of control is affected by calling a function
  - With a function call, we pass some parameters
  - These parameters are used within the function
  - A value is computed
  - The value is returned to the function which initiated the call
  - A function could call itself, these are called *recursive function* calls
- Function prototype, function definition, and function call

Passing Arguments to Functions

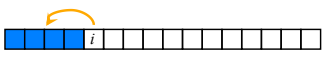
- In C, function arguments are passed "by value"
  - values of the arguments given to the called function in temporary variables rather than the originals
  - the modifications to the parameter variables do not affect the variables in the calling function
- "Call by reference"
  - variables are passed by reference
    - subject to modification by the function
  - achieved by passing the "address of" variables

Selection Vs Insertion Sort

- Scanning from left to right
- Selection sort



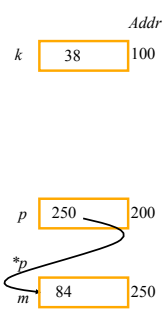
  - Swaps the i^{th} element with the largest unsorted element
- Insertion sort



  - Inserts the ith element into its proper place

What is a Pointer?

- *Recap*: a variable ***int*** *k*
  - Names a memory location that can hold one value at a time
  - Memory is allocated statically at compile time
  - One name ⇔ one value
- A pointer variable ***int*** ***p***
  - Contains the address of a memory location that contains the actual value
  - Memory can be allocated at runtime
  - One name ⇔ many values



l-value and *r*-value

- Given a variable *k*
  - Its *l*-value refers to the address of the memory location
  - *l*-value is used on the left side of an assignment
    - Ex. *k* = expression
  - Its *r*-value refers to the value stored in the memory location
  - *r*-value is used in the right hand side of an assignment
    - Ex. *var* = *k* + …
- Pointers allow one to manipulate the *l*-value!

Accessing Arrays Using Pointers

- The name of the array is the address of the first element in the array
- In C, we can replace

  ptr = &myArray[0];

  with

  ptr = myArray;

  to achieve the same result

Strings

- A string is a array of characters terminated by the null character, '\0'
- A string is written in double quotes
  - Example: "This is a string"
- " " – empty string
- Anything within single quotes gets a number associated with it
- 'This is rejected by the C Compiler'
- Character arrays can also be accessed by pointers

Strcpy Using Pointers

```
char *myStrcpy(char *destination, char *source)
{
  char *p = destination;
  while (*source != '\0')
    {
      *p++ = *source++;
    }
  *p = '\0';
  return destination;
}
```

Ted Jenson's tutorial on pointers
http://pweb.netcom.com/~tjensen/ptr/cpoint.htm