Multi-Dimensional Arrays

Last updated 8/16/23

These slides introduce multi-dimensional arrays

Multi-Dimensional Arrays

- 2 Dimensional Arrays

  Consider a table

1	2	3	4	5
6	5	4	3	2
12	11	13	14	15
19	17	16	3	1

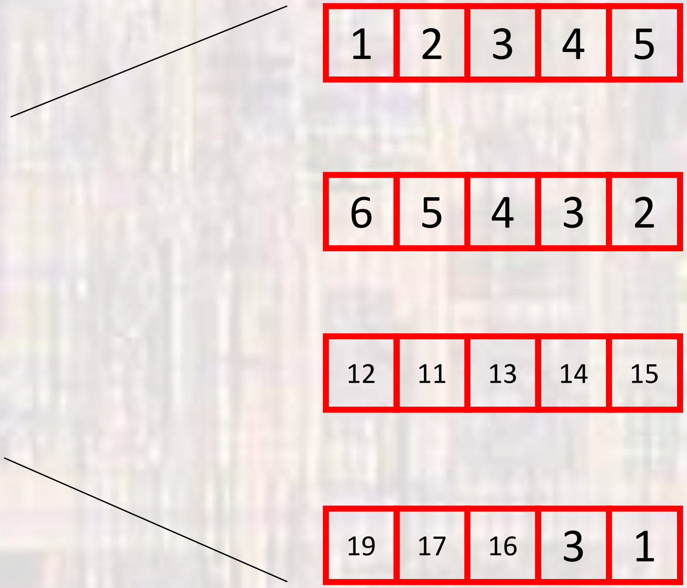
4 rows x 5 columns

Multi-Dimensional Arrays

- 2 Dimensional Arrays

  Consider a table

1	2	3	4	5
6	5	4	3	2
12	11	13	14	15
19	17	16	3	1



1 2 3 4 5

6 5 4 3 2

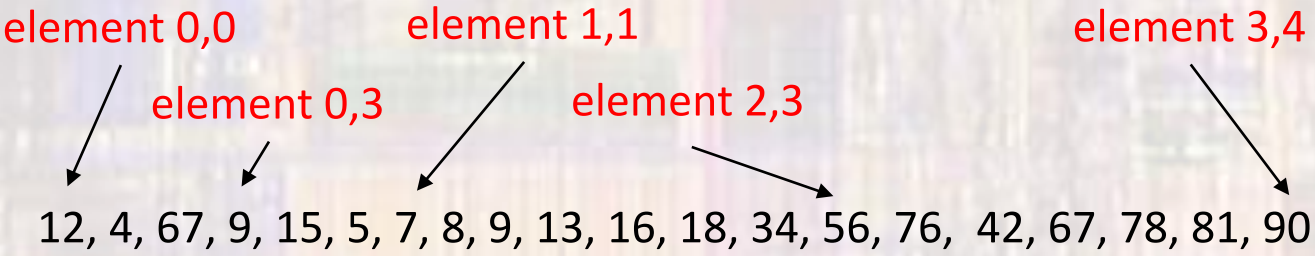
12 11 13 14 15

19 17 16 3 1

4 – 1 Dimensional Arrays

Multi-Dimensional Arrays

- 2 Dimensional Arrays
  - First element [0][0] is upper-left most element



element 0,0

element 0,3

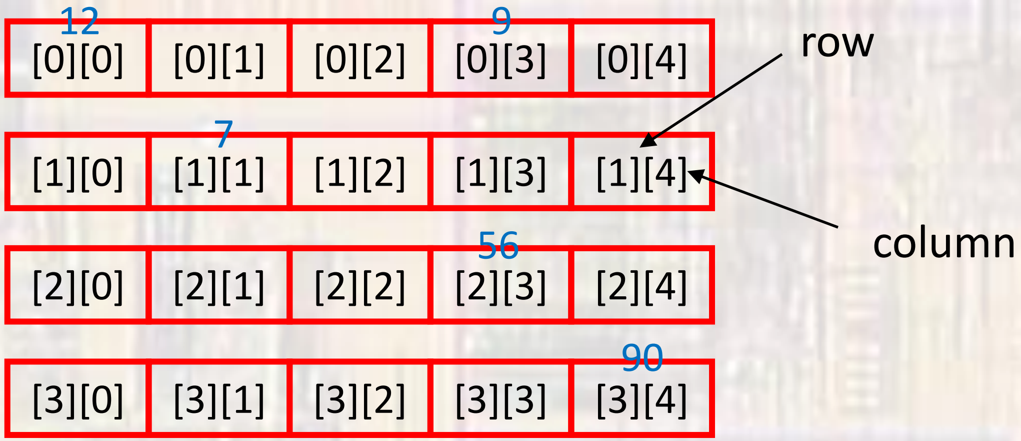
element 1,1

element 2,3

element 3,4

12, 4, 67, 9, 15, 5, 7, 8, 9, 13, 16, 18, 34, 56, 76, 42, 67, 78, 81, 90

- Array of Arrays – 4x5 – 4 rows by 5 columns
  - Indices are ROW-COL format



12 [0][0]	[0][1]	[0][2]	9 [0][3]	[0][4]
[1][0]	7 [1][1]	[1][2]	[1][3]	[1][4]
[2][0]	[2][1]	[2][2]	56 [2][3]	[2][4]
[3][0]	[3][1]	[3][2]	[3][3]	90 [3][4]

row

column

Multi-Dimensional Arrays

- Declaration

```
type arrayName[#rows][#cols];
```

Fixed size array – size known during compilation

```
int scores[4][5];
char first_name[15][20];
```

Variable size array – size only known during execution

```
float testAve[classSize][numTests];
int numAs[gradesGE90][numClasses];
```

where classSize,gradesGE90, numTests, numClasses
are integral variables

Multi-Dimensional Arrays

- Declaration
  - Un-initialized arrays contain garbage

```
type arrayName[#rows][#cols];

int myArray[3][4];
```

?	?	?	?	?
?	?	?	?	?
?	?	?	?	?
?	?	?	?	?

Multi-Dimensional Arrays

- Initialization

```
type arrayName[#rows][#cols] = {comma separated list};

int myArray[3][4] = {1,2,3,4,1,2,3,4,1,2,3,4};       // basic

int myArray[3][4] = {
                      {1,2,3,4},
                      {1,2,3,4},
                      {1,2,3,4}
                     };                       // preferred

int myArray[3][4] = {0};                      // all zeros
```

Multi-Dimensional Arrays

- Variable length arrays

Variable length arrays **cannot** have an initialization

```
float testAve[classSize][numTests];
int numAs[gradesGE90][numClasses];
```

Multi-Dimensional Arrays

- Accessing elements

```
foo = myArray[1][2];        // foo = 4
foo = myArray[2][foo];      // foo = 15
```

myArray

1	2	3	4	5
6	5	4	3	2
12	11	13	14	15
19	17	16	3	1

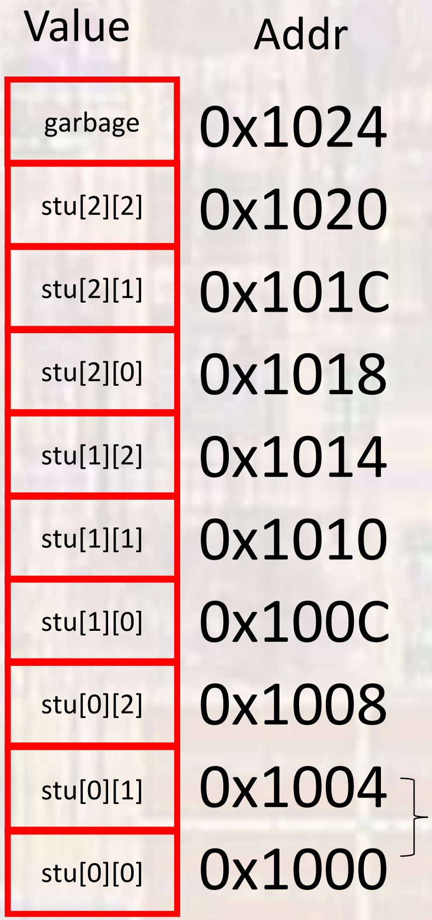
```
myArray[0][0] = 0;
```

```
foo = 1;
myArray[foo + 1][foo + 2] = 6;
```

0	2	3	4	5
6	5	4	3	2
12	11	13	6	15
19	17	16	3	1

Multi-Dimensional Arrays

- Memory Structure
  - Name is actually the address of the beginning of the array (a pointer)
  - Index is the offset from the name address
    - not an address
  - NxM array → linear in memory



Value	Addr
garbage	0x1024
stu[2][2]	0x1020
stu[2][1]	0x101C
stu[2][0]	0x1018
stu[1][2]	0x1014
stu[1][1]	0x1010
stu[1][0]	0x100C
stu[0][2]	0x1008
stu[0][1]	0x1004
stu[0][0]	0x1000

addr offset in memory is calculated by the compiler to match the size of the element types

offset = size_of_type * (row * #cols + col)

Multi-Dimensional Arrays

- Index Range Checking
  - C does NOT check array index ranges

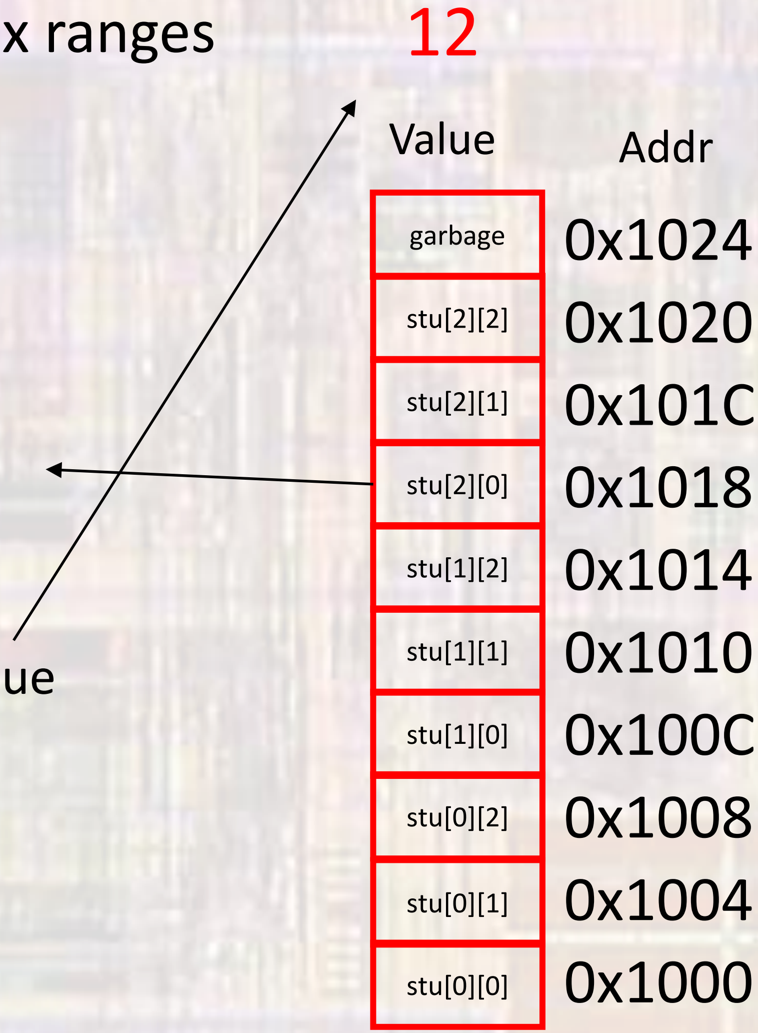
```
int stu[3][3];
…

foo = stu[1][3];
```

sets foo = stu[2][0] wrong

```
stu[3][2] = 12;
```

overwrites critical data value



12

Value	Addr
garbage	0x1024
stu[2][2]	0x1020
stu[2][1]	0x101C
stu[2][0]	0x1018
stu[1][2]	0x1014
stu[1][1]	0x1010
stu[1][0]	0x100C
stu[0][2]	0x1008
stu[0][1]	0x1004
stu[0][0]	0x1000

Multi-Dimensional Arrays

- Dimensions Beyond 2
  - All the same rules apply
    - Linear in memory
    - No bounds checking
    - Name is pointer
    - Index is offset
  - Difficult to visualize

```
int myArray[3][7][2][5];
```