CSC111

Pseudo Code – assignment tip

- Plain language description of the steps of an algorithm
- A good way to start implementing an algorithm
- Explains what needs to be done in the program
- Uses standard programming structures

Debugging

- Don’t assume things work the way they’re meant to:
  - Determine what the output of the code should be, then manually execute the code and trace the variables.
  - Insert print statements in strategic places in your code to print out/track variables or to see if you reach that spot in the code
    - Remember you need to remove these!
  - Comment out code – reduce the problem area
  - Fix one issue at a time. Then recompile and test (and test, test, test)…

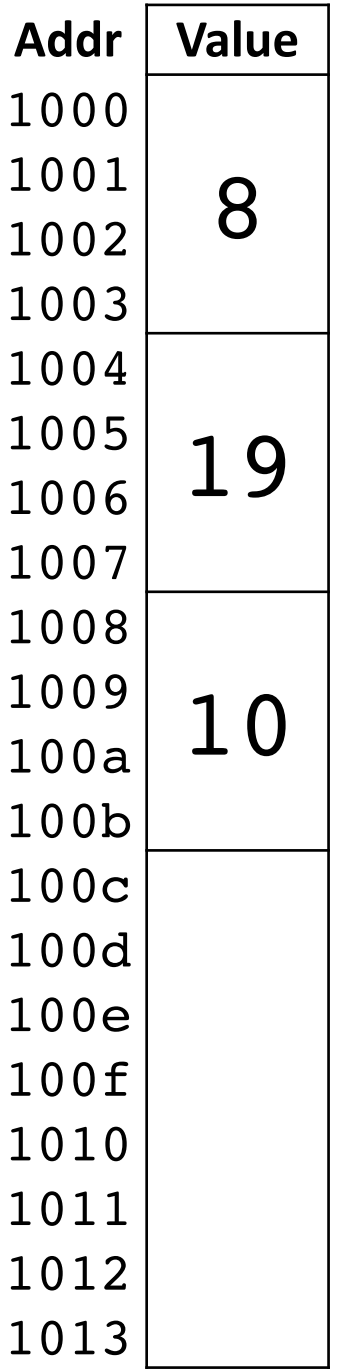
Remember: Arrays are pointers…

```
int my_ints[3] = {8,19,10};
```

`*my_ints` evaluates to `8`

`*(my_ints+1)` evaluates to `19`

`*(my_ints+2)` evaluates to `10`



2D Arrays and Arrays of pointers

Declaring an array of a specified size:

```
int my_ints[10];
```

Declaring a 2D array of a specified size:

```
int my_ints[2][3];
```

Allocates space in memory with initial **garbage** values

Declaring AND initializing an array of a specified size:

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

Declaring AND initializing a 2D array of a specified size:

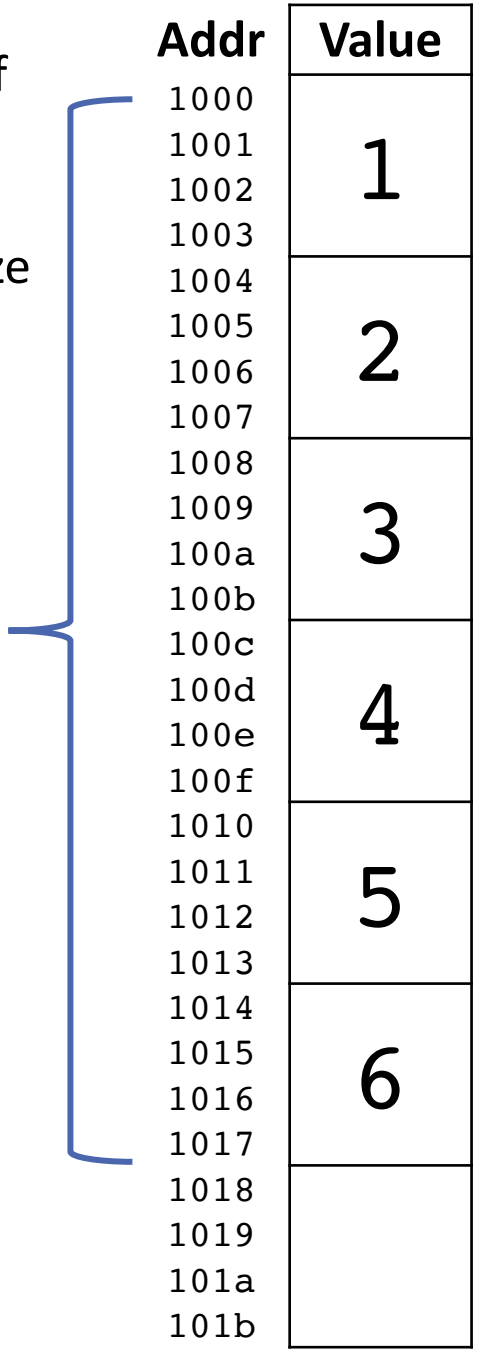
```
int my_ints[2][3] = {{1, 2, 3},
                     {4, 5, 6}};
```

Allocates space in memory with initial **given** values

2D Arrays in memory…

- A contiguous block of memory is allocated
- space for values of specified type and size
- 2*3 integers here

```
int my_ints[2][3] = {{1, 2, 3},
                     {4, 5, 6}};
```

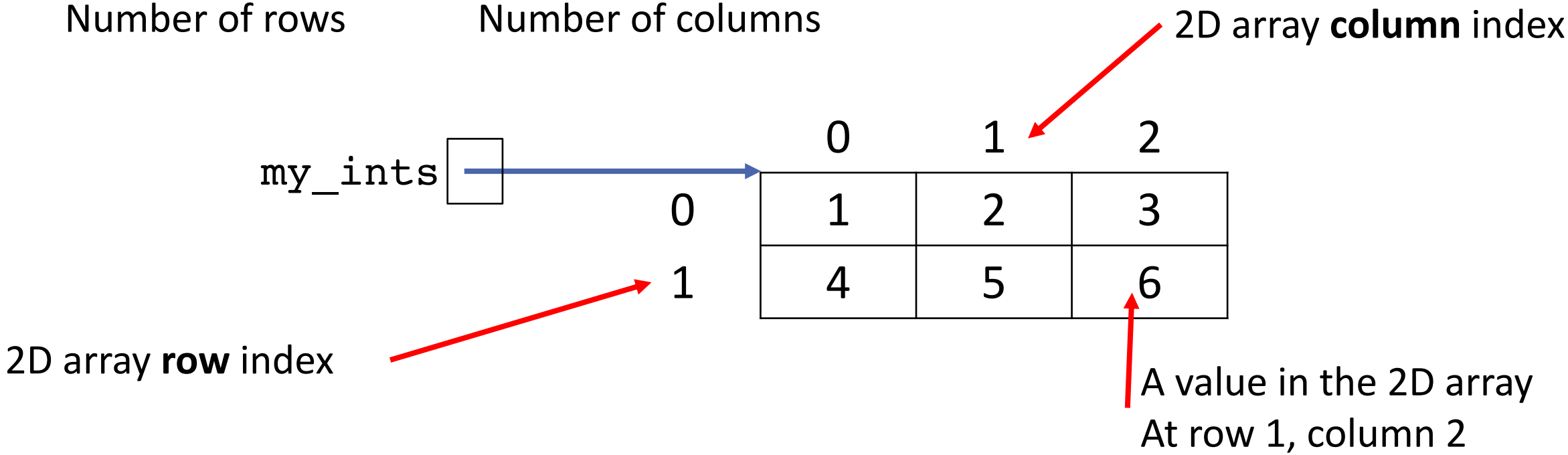


Visualizing 2D arrays…

```
int my_ints[2][3] = {{1, 2, 3},
                     {4, 5, 6}};
```

Number of rows

Number of columns



2D arrays - accessing and assigning values

```
int my_ints[2][3] = {{1, 2, 3},
                     {4, 5, 6}};
printf("value at row 0, column 0: %d\n", my_ints[0][0]);
printf("value at row 0, column 1: %d\n", my_ints[0][1]);
printf("value at row 0, column 2: %d\n", my_ints[0][2]);
printf("value at row 1, column 0: %d\n", my_ints[1][0]);
printf("value at row 1, column 1: %d\n", my_ints[1][1]);
printf("value at row 1, column 2: %d\n", my_ints[0][2]);

int sum = my_ints[0][1] + my_ints[1][2];
my_ints[1][1] = sum;
my_ints[1][0]++;
```

Using a nested for-loop to traverse 2D array

```
int my_ints[2][3] = {{1, 2, 3}, {4, 5, 6}};

//instead of duplicating code:
printf("value at row 0, column 0: %d\n", my_ints[0][0]);
printf("value at row 0, column 1: %d\n", my_ints[0][1]);
printf("value at row 0, column 2: %d\n", my_ints[0][2]);
printf("value at row 1, column 0: %d\n", my_ints[1][0]);
printf("value at row 1, column 1: %d\n", my_ints[1][1]);
printf("value at row 1, column 2: %d\n", my_ints[1][2]);


//put the repeating code within the body of a loop:
int row, col;
int num_rows = 2, num_cols = 3;
for(row=0; row<num_rows; row++) {
    for(col=0; col<num_cols; col++) {
        printf("value at row %d:, col %d:  %d\n", row, col, my_ints[row][col]);
    }
}
```

Demo

rows in a 2D array are 1D arrays

`my_ints[0]` is a 1D array, therefore…

`my_ints[0]` is compatible with an `int*`
`my_ints[1]` is compatible with an `int*`

dereferencing `my_ints` at a specific [row], will get the **value** of the first element in that row

This can be done using the dereference operator or the square bracket operator:

```
printf("%d",  *my_ints[0]);
  1
printf("%d",  my_ints[0][0]);
  1

printf("%d",  *my_ints[1]);
  4
printf("%d",  my_ints[1][0]);
  4
```

Dereferencing and pointer arithmetic

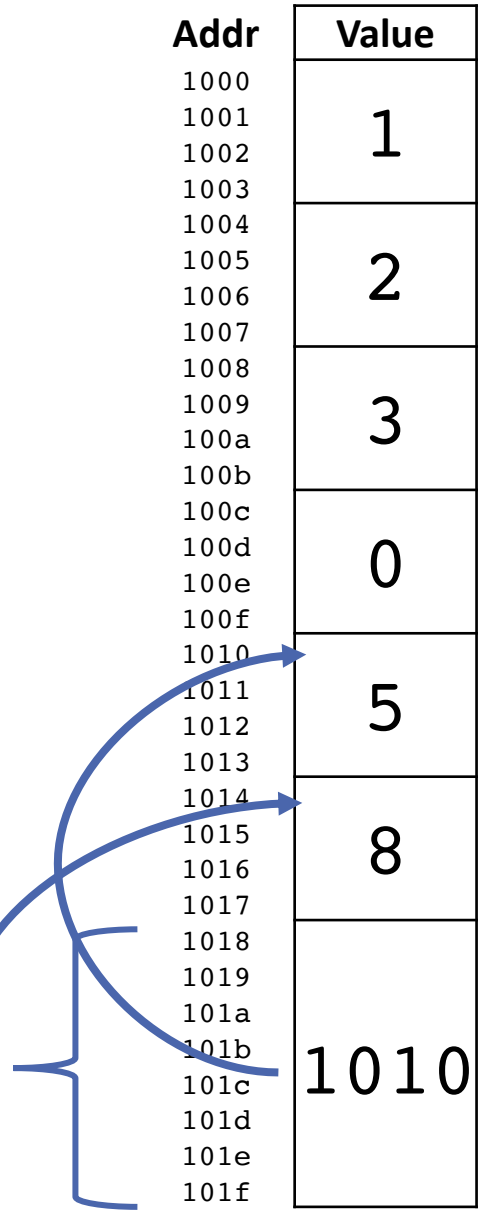
```
int my_ints[2][3] = {{1, 2, 3},
                     {0, 5, 8}};
```

`int* ptr = my_ints[1] + 1;`

```
= address of the second row + sizeof an int
= 100c + sizeof an int
= 100c + 4
= 1010
```

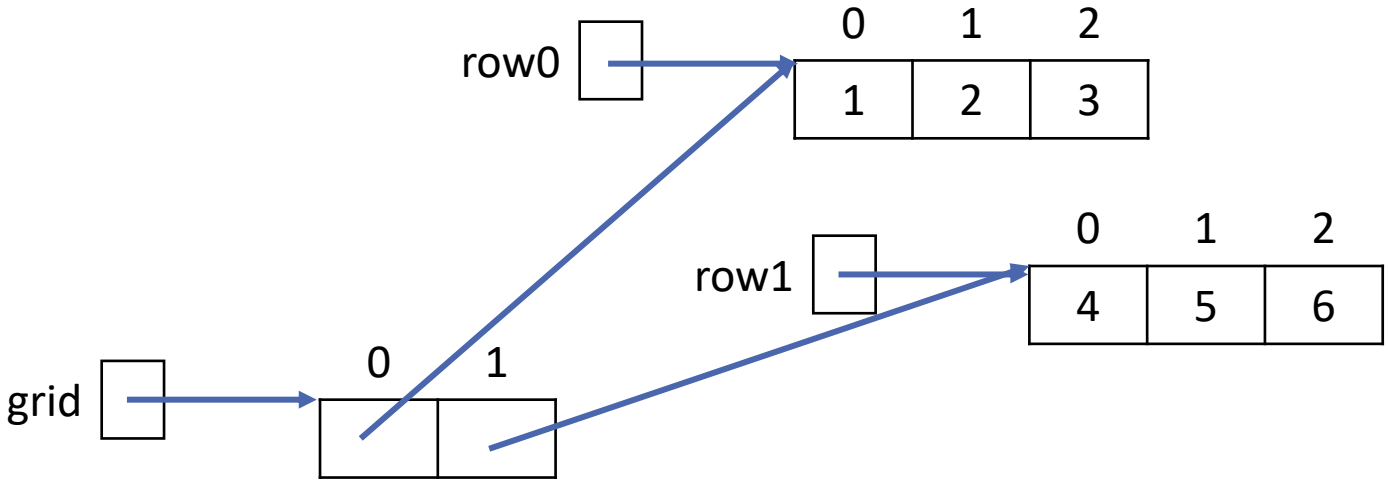
```
printf("%p", ptr);
      1010
printf("%d", *ptr);
      5
printf("%d", *(ptr + 1));
      8
```

```
*(ptr + 1)
*(ptr + sizeof an int)
*(1010 + 4)
*1014
```



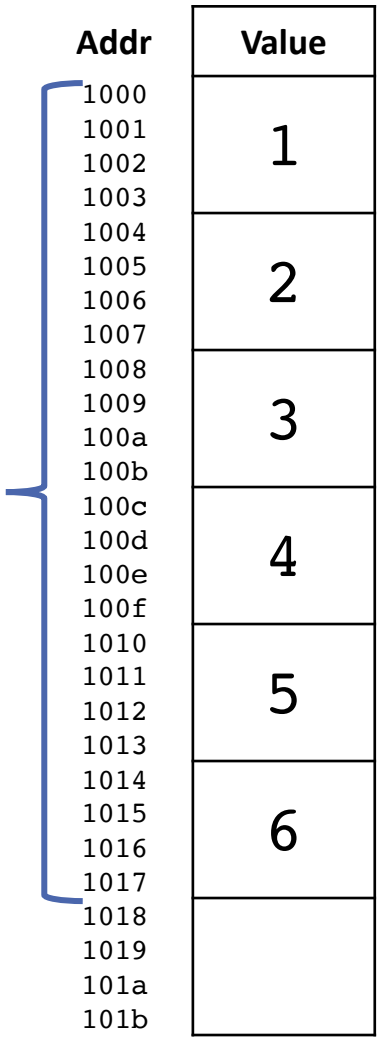
Arrays of integer pointers (arrays of arrays)

```
int row0[3] = {1, 2, 3};
int row1[3] = {4, 5, 6};
int* grid[2] =  {row0, row1};
```

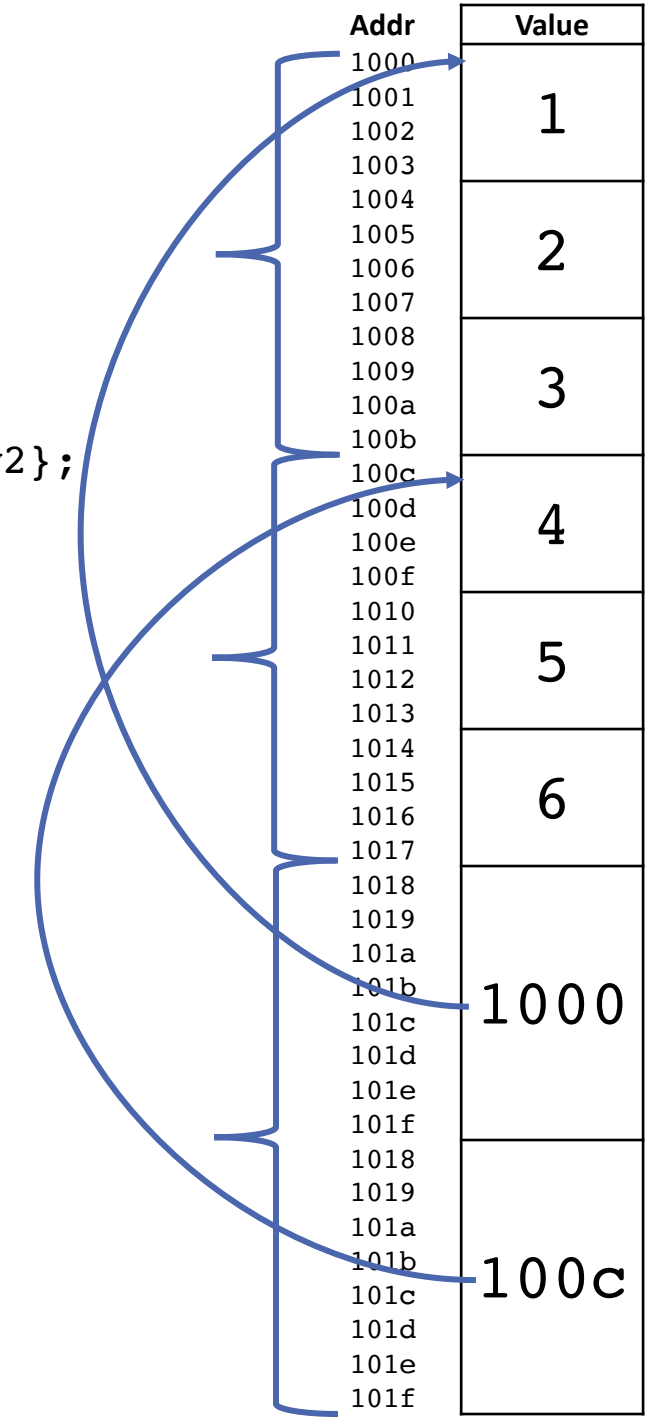


memory allocation differs

```
int my_ints[2][3] = {{1, 2, 3},
                     {4, 5, 6}};
```




```
int row1[3] = {1, 2, 3};
int row2[3] = {4, 5, 6};
int* grid[2] = {row1, row2};
```



accessing elements is the same

```
int table[2][3] = {{1, 2, 3},
                   {4, 5, 6}};


int row, col;
for(row=0; row<2; row++) {
    for(col=0; col<3; col++) {
       printf("%d ", table[row][col]);
    }
}
```

```
int a1[3] = {1, 2, 3};
int a2[3] = {4, 5, 6};
int* grid[2] = {a1, a2};

int row, col;
for(row=0; row<2; row++) {
    for(col=0; col<3; col++) {
       printf("%d ", grid[row][col]);
    }
}
```

Passing 2D arrays to Functions

- Needs to know the columns in 2D array
- Should be passed the rows in array

```c
#include <stdio.h>
#define MAXCOLS 10

void print_table(int table[][MAXCOLS], int num_rows, int num_cols);
//main omitted intentionally

/* Purpose: print values in table with dimensions num_rows by num_cols
 * Parameters: int table[][MAXCOLS], 2d array of integers
 *             int num_rows, number of rows in table, >=0
 *             int num_cols, number of columns in table, >=0 and <= MAXCOLS
 */
void print_table (int table[][MAXCOLS], int num_rows , int num_cols) {
        int row, col;

        for(row=0; row<num_rows; row++) {
                for(col=0; col<num_cols; col++) {
                        printf("%d ", table[row][col]);
                }
                printf("\n");
        }
}
```

LIMITATION:
Allocating more memory than necessary
Cannot be wider than MAXCOLS

```c
#include <stdio.h>

void print_table_ptrs(int* table[], int num_rows, int num_cols);
//main omitted intentionally

/* Purpose: print values in table with dimensions num_rows by num_cols
 * Parameters: int* table[], array of integer arrays
 *             int num_rows, number of rows in table, >=0
 *             int num_cols, number of columns in table, >=0
 */
void print_table_ptrs(int* table[], int num_rows, int num_cols) {
        int row, col;

        for(row=0; row<num_rows; row++) {
                for(col=0; col<num_cols; col++) {
                        printf("%d ", table[row][col]);
                }
                printf("\n");
        }
}
```

LIMITATION:
Data is not allocated in a contiguous block of memory which can impact performance of data access.

```
#include <stdio.h>
#define MAXCOLS 10

void print_table(int table[][MAXCOLS], int num_rows, int num_cols);
void print_table_ptrs(int* table[], int num_rows, int num_cols);

int main ( ) {
        int table_2d[2][MAXCOLS] = {{10, 20, 30, 40},
                                    {60, 70, 80, 90}};

        int a1[3] = {1, 2, 3};
        int a2[3] = {4, 5, 6};
        int* table_ptrs[2] = {a1, a2};

        print_table(table_2d, 2, 4);
        print_table_ptrs(table_ptrs, 2, 3);

        print_table(table_ptrs, 2, 4);
        print_table_ptrs(table_2d, 2, 3);

        return 0;
}
```

```c
#include <stdio.h>

void print_table_ptrs(int num_rows, int num_cols, int table[num_rows][num_cols]);
//main omitted intentionally

/* Purpose: print values in table with dimensions num_rows by num_cols
 * Parameters:   int num_rows, number of rows in table, >=0
 *               int num_cols, number of columns in table, >=0
 *               int table[num_rows][num_cols]
 */
void print_table_ptrs(int num_rows, int num_cols, int table[num_rows][num_cols]) {
        int row, col;

        for(row=0; row<num_rows; row++) {
                for(col=0; col<num_cols; col++) {
                        printf("%d ", table[row][col]);
                }
                printf("\n");
        }
}
```

Demo – functions

What is the output of the following Code?

```
int main(){
     int a[3][2]
}

int mystery(int row, int col, int ar[row][col] ) {
     int sum =0;
     for(int i=0; i<row; i++){
          for(int j=0; j<col; j++){
          }
     }
}
```

Write a function…

- Write a function sum_matrices that takes five arguments: three 2D arrays no larger than 2 x 2.
- Assume each of the 2D arrays is a matrix, each with the same dimensions(*m* rows by *n* columns).
  You are to add the first and second matrix together, storing the result in the third matrix.
- Here is an example of two matrices being added to produce the answer on the right:

2	-3		1	4		3	1
		+			=		
-7	5		0	2		-7	7