CS 354 - Machine Organization & Programming
Tuesday Feb 6, and Thursday Feb 8, 2024

Submit Exam Conflicts and Accommodations Requests Today

PM BYOL #2: Vim, SCP, GDB

Project p2A: Due on or before 2/16

Project p2B: Due on or before 2/23 (due after E1, but should be written before E1)

Homework hw1 DUE: Monday Feb 12, must first mark hw policies page

Homework hw2 DUE: Monday Feb 19, must first mark hw policies

Week 3 Learning Objectives (at a minimum be able to)

- use <string.h> functions: strlen, strcp, strncpy, strcat, on C strings
- use information passed in via command line arguments CLAs in program
- understand and show binary representation and byte ordering for pointers and arrays
- create, allocate, and fill 2D arrays on heap
- create, allocate, and fill 2D arrays on the stack
- diagram 2D arrays on stack and on heap
- understand and show byte representation of elements in 2D arrays
- understand and use struct to create compound variables with different typed values
- next compound types within other compound types
- pass structs to and return them from functions
- pass addresses to structs

This Week

Tuesday	Thursday
Meet C strings and string.h (from last week) Command-line Arguments Recall 2D Arrays 2D Arrays on the Heap 2D Arrays on the Stack 2D Arrays: Stack vs. Heap	Array Caveats Meet Structures Nesting in Structures and Arrays of Structures Passing Structures Pointers to Structures
Read before next Week K&R Ch. 7.1: Standard I/O K&R Ch. 7.2: Formatted Output - Printf K&R Ch. 7.4: Formatted Input - Scanf K&R Ch. 7.5: File Access Read before next week Thursday B&O 9.1 Physical and Virtual Addressing B&O 9.2 Address Spaces B&O 9.9 Dynamic Memory Allocation B&O 9.9.1 The malloc and free Functions **Do:** Work on project p2A / Start project p2B, and finish homework hw1 (arrays and pointers)	

Command Line Arguments

What? *Command line arguments* are a whitespace separated list of input entered after the terminal's command prompt on command line

program arguments: Args that follow command or program name

Cmd

```
$gcc myprog.c -Wall -m32 -std=gnu99 -o myprog
```

6 arguments

Why?

enables info to be passed to prog when it begins.

How?

char ** argv

```
int main(int argc, char *argv[]) {
   for (int i = 0; i < argc; i++)
      printf("%s\n", argv[i]);
   return 0;
}
```

array of arrray of char

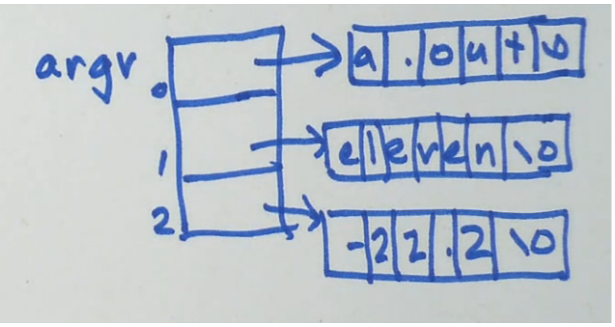
argc:

argument count, #CLA

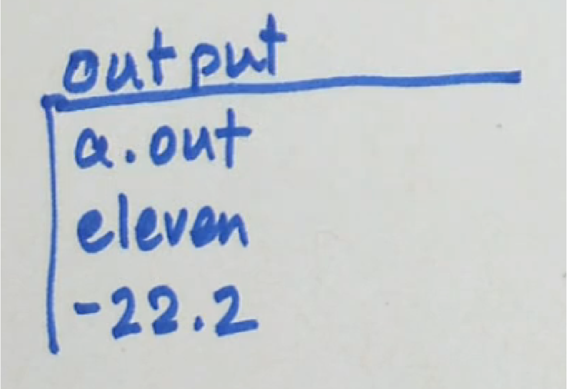
argv:

argument vector, array of CLA

→ Assume the program above is run with the command "`$a.out eleven -22.2`"
Draw the memory diagram for argv.



➢ Now show what is output by the program:



output

```
a.out
eleven
-22.2
```

Recall 2D Arrays

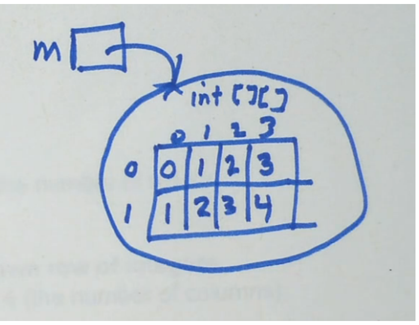
2D Arrays in Java

row

```
int[][] m = new int[2][4];
```

column

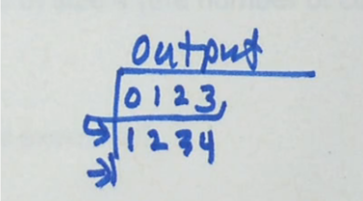
→ Draw a basic memory diagram of resulting 2D array:



```
for (int i = 0; i < 2; i++)
   for (int j = 0; j < 4; j++)
      m[i][j] = i + j;
```

➢ What is output by this code fragment?

```
for (int i = 0; i < 2; i++) {
   for (int j = 0; j < 4; j++)
      printf("%i", m[i][j]);
   printf("\n");
}
```



Output

```
0123
1234
```

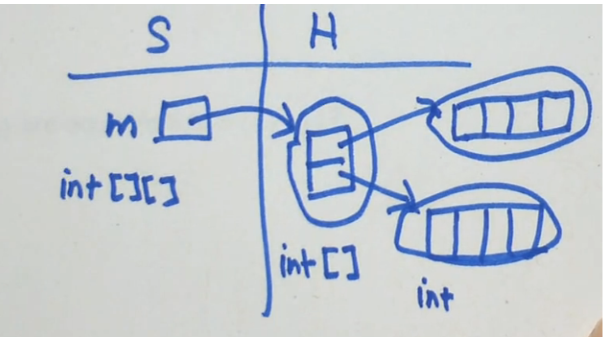
→ What memory segment does Java use to allocate 2D arrays?

HEAP

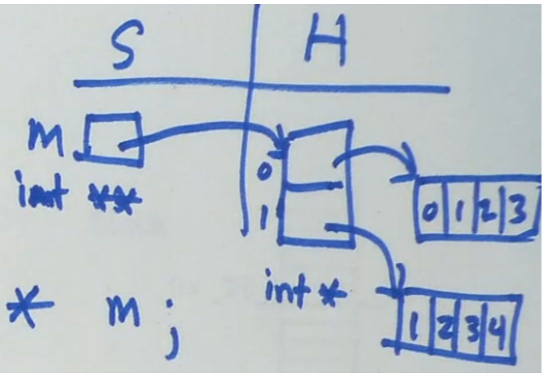
→ What technique does Java use to layout a 2D array?

1D Array of 1D Arrays

→ What does the memory allocation look like for `m` as declared at the top of the page?



2D Arrays on the Heap



2D “Array of Arrays” in C

→ **1. Make a 2D array pointer named `m`.**
Declare a pointer to an integer pointer. int ** m;

→ **2. Assign `m` an “array of arrays”.**
Allocate of a 1D array of integer pointers of size 2 (the number of rows).

m = malloc (sizeof(int *)*2);
if(m == null) ... (check to see if something fucked up)

→ **3. Assign each element in the “array of arrays” it own row of integers.**
Allocate for each row a 1D array of integers of size 4 (the number of columns).

*(m+0) = malloc(sizeof(int) *4);
*(m+1) = malloc(sizeof(int) *4);

➢ What is the contents of `m` after the code below executes?

```
for (int i = 0; i < 2; i++) {
   for (int j = 0; j < 4; j++)
      m[i][j] = i + j;
```

To Get Values from 2D Array: (3) to (-1)

((m+1)+2) = -1;

→ Write the code to free the heap allocated 2D array.

free(*(m+1));
free(*m);
free(m);
m = NULL;

❈ *Avoid memory leaks; free the components of your heap 2D array*
in reverse order of allocation:

Address Arithmetic

→ Which of the following are equivalent to `m[i][j]`?

a.) `*(m[i]+j)` ok
b.) `(*(m+i))[j]`ok
c.) `*(*(m+i)+j)` ok. use in p2a and p2b

❈ *`m[i][j]`* == *(*(m+i)+j)

compute row `i`‘s address m+1
dereference address in 1. gives *()
compute element `j`‘s address in row `i` +j
dereference the address in 3. to access element at row `i` column `j` *()

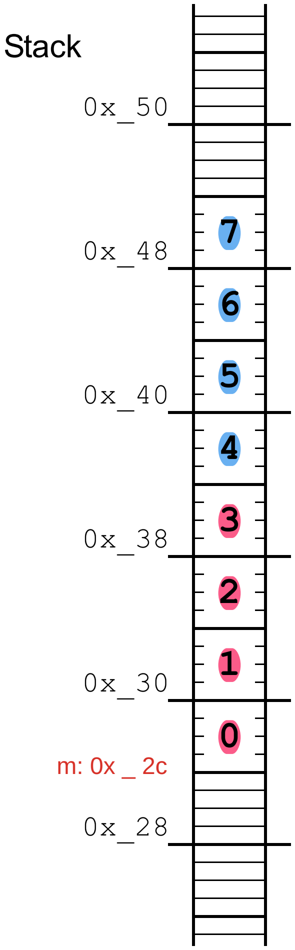
❈ *`m[0][0]`* == **m

2D Arrays on the Stack

Stack Allocated 2D Arrays in C

```
void someFunction(){
   int m[2][4] = {{0,1,2,3},{4,5,6,7}};
```

SAA Initializer list



✵ *2D arrays allocated on the stack*

are laid out in row-major order, as a single continuous block of memory with one row after another

Stack & Heap 2D Array Compatibility

→ For each one below, what is provided when used as a source operand? What is its type and scale factor?

1. `**m`? == *(*(m)) == m[0][0]

   type? int
   scale factor? none

2. `*m`? `*(m+i)`? addr of row i

   type? int *
   scale factor? (how do we skip to row i?)
   STACK: sizeof(int) * COLS, eg 16 bytes
   HEAP: sizeof(int *), eg 4 bytes

3. `m[0]`? `m[i]`? same as 2.

4. `m`? STACK: addr of start of 2D array
   HEAP: address of start of 1D array with addresses to other 1D array

   type? int **
   scale factor?

   to skip to "next" element
   STACK: sizeof(element)
   HEAP:

For 2D STACK Arrays ONLY

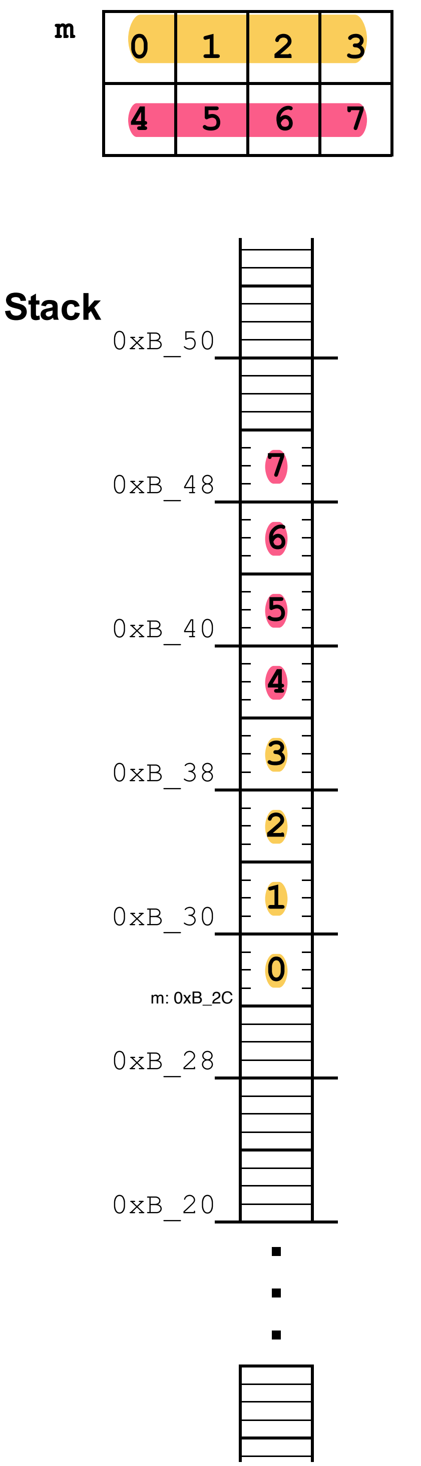
✵ *m* and **m* are same address but not same type

✵ *m[i][j]* == *(*(m+i)+j) == *(*m+ cols * i +j)

Stack and Heap

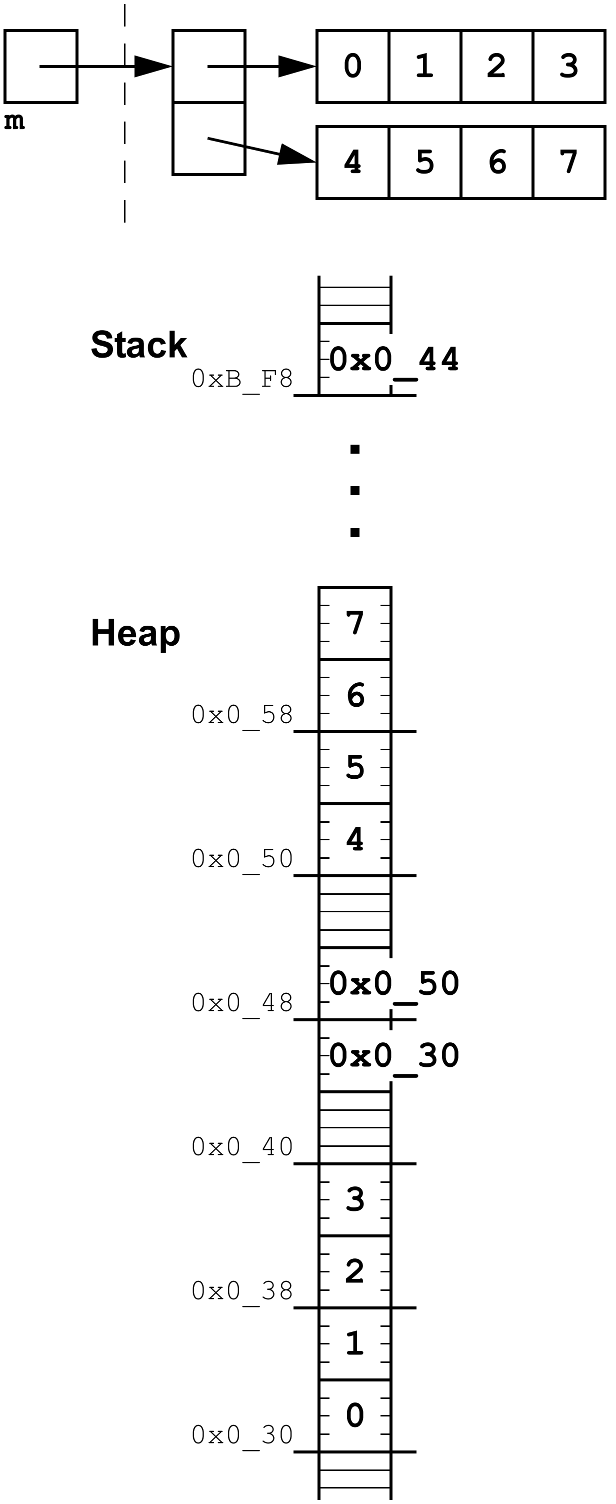
2D Arrays: Stack vs. Heap

Stack: row-major order layout



m[i][j] = *(*(m+i)+j)
SAA: *(*m + cols*i +j)

Heap: array-of-arrays layout



Array Caveats

- *Arrays have no bounds checking!*

```
int a[5]; // SAA
for (int i = 0; i < 11; i++)
   a[i] = 0;
```

Buffer Overflow

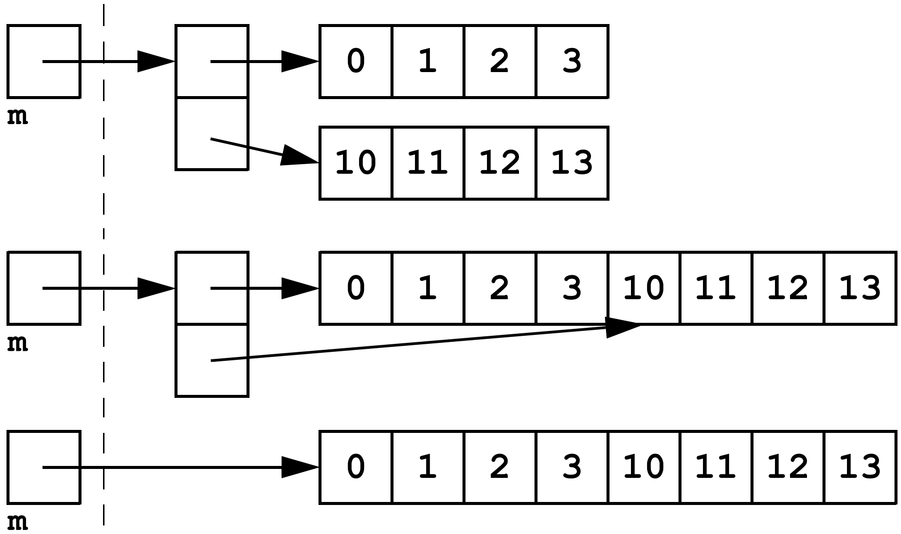
- *Arrays cannot be return types!*

```
int[] makeIntArray(int size) {
   return malloc(sizeof(int) * size);
}
```

- *Not all 2D arrays are alike!*

  → What is the layout for ALL 2D arrays on the stack?

  → What is the layout for 2D arrays on the heap?



- *An array argument must match its parameter's type!*

- *Stack allocated arrays require all but their first dimension specified!*

```
int a[2][4] = {{1,2,3,4},{5,6,7,8}};
printIntArray(a,2,4); //size of 2D array must be passed in (last 2 arguments)
```

  → Which of the following are type compatible with `a` declared above?

```
void printIntArray(int a[2][4],int rows,int cols)
void printIntArray(int a[8][4],int rows,int cols)
void printIntArray(int a[][4], int rows,int cols)
void printIntArray(int a[4][8],int rows,int cols)
void printIntArray(int a[][],  int rows,int cols)
void printIntArray(int (*a)[4],int rows,int cols)
void printIntArray(int **a,    int rows,int cols)
```

  → Why is all but the first dimension needed?

Meet Structures

What? A *structure*

- A user defined type
- A compound unit of storage with data members of different types
- Access using identifier and data member name
- Allocated as a continuous fixed sized block of memory

Why?

How? Definition

```
struct <typename> {            typedef struct {
   <data-member-declaratns>;      <data-member-declaratns>;
};                             } <typename>;
```

→ Define a structure representing a date having integers month, day of month, and year.

How? Declaration

→ Create a `Date` variable containing today's date.

dot operator:

❈ *A structure's data members*

❈ *A structure's identifier used as a source operand*

❈ *A structure's identifier used as a destination operand*

```
struct Date tomorrow;
tomorrow = today;
```

Nesting in Structures and Array of Structures

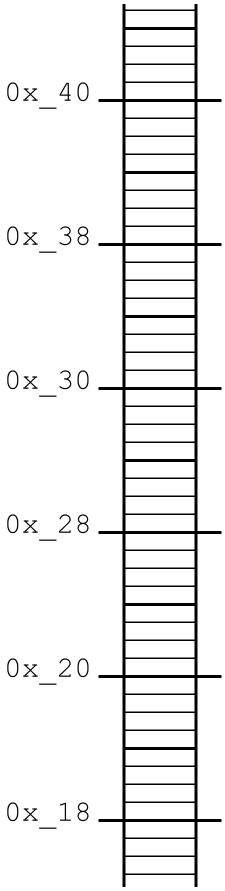
Nesting in Structures

→ Add a `Date` struct, named `caught`, to the structure code below.

```
typedef struct { ... } Date; //assume as done on prior page

typedef struct {
   char  name[12];
   char  type[12];
   float weight;
   Date caught;

} Pokemon;
```



❋ *Structures can contain*
other structs and arrays nested as deeply as you wish

→ Identify how a `Pokemon` is laid out in the memory diagram.

Array of Structures

❋ *Arrays can have*

→ Statically allocate an array, named `pokedex`,
and initialize it with two pokemon.

→ Write the code to change the weight to 22.2 for the Pokemon at index 1.

→ Write the code to change the month to 11 for the Pokemon at index 0.

Passing Structures

→ Complete the function below so that it displays a `Date` structure.

```
void printDate (Date date) {
        printf("%i/%2i/%4i \n", date.month, date.day, date.year);


}
```

✵ *Structures are passed-by-value to a function,*

Consider the additional code:

```
//assume code for Date, Pokemon, printDate same as prior pages

void printPm(Pokemon pm) {
   printf("\nPokemon Name      : %s",pm.name);
   printf("\nPokemon Type      : %s",pm.type);
   printf("\nPokemon Weight    : %f",pm.weight);
   printf("\nPokemon Caught on : "); printDate(pm.caught);
   printf("\n");
}

int main(void) {
   Pokemon pm1 = {"Abra","Psychic",30,{1,21,2017}};
   printPm(pm1);
   ...
```

→ Complete the function below so that it displays a `pokedex`.

```
void printDex(Pokemon dex[], int size) {
```

✵ *Recall: Arrays are passed-by-value to a function,*

Pointers to Structures

Why? Using pointers to structures

- Avoid copying overhead of pass by value
- Allows func to change struct data numbers
- Enables heap allocation of structs
- Enables linked structs

How?

→ Declare a pointer to a `Pokemon` and dynamically allocate it’s structure.

→ Assign a weight to the `Pokemon`.

points-to operator:

→ Assign a name and type to the `Pokemon`.

→ Assign a caught date to the `Pokemon`.

→ Deallocate the `Pokemon`‘s memory.

→ Update the code below to efficiently pass and print a Pokemon.

```
void printPm(Pokemon    pm) {
   printf("\nPokemon Name      : %s",pm   name);
   printf("\nPokemon Type      : %s",pm   type);
   printf("\nPokemon Weight    : %f",pm   weight);
   printf("\nPokemon Caught on : "); printDate(pm   caught);
   printf("\n");
}
int main(void) {
   Pokemon pm1 = {"Abra","Psychic",30,{1,21,2017}};
   printPm(  pm1 )
```