CS 354 - Machine Organization & Programming
Tuesday Sept 17, and Thursday Sept 19, 2024

Submit Exam Conflicts and Accommodations Requests Today

PM BYOL #2: Vim, SCP, GDB

Project p2A: Due on or before 9/27

Project p2B: Due on or before 10/4 (due after E1, but should be written before E1)

Homework hw1 DUE: Monday 9/23, must first mark hw policies page

Homework hw2 DUE: Monday 9/30, 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
(finish 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
    <u>B&O</u> 9.1 Physical and Virtual Addressing
    <u>B&O</u> 9.2 Address Spaces
    <u>B&O</u> 9.9 Dynamic Memory Allocation
    <u>B&O</u> 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

> *program arguments*:follow the program/command name and are typically file names and/or options/flags that specify the program's behavior.

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

Why?

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

argc:argument count
        number of the cla's

argv:argument values/vector
        an array of char pointers where each element points to a cla string in order entered

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

        NOTE: these are all C strings (not int or other types)

➢ Show what is output by the program:

Recall 2D Arrays

2D Arrays in Java

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

→ 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");
}
```

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

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

→ 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.

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

→ **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).

➢ 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;
```

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

❋ *Avoid memory leaks; free the components of your heap 2D array*

Address Arithmetic

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

   a.) `*(m[i]+j)`
   b.) `(*(m+i))[j]`
   c.) `*(*(m+i)+j)`

❋ *m[i][j]*

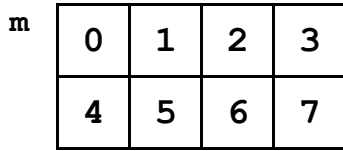   compute row `i`'s address
   dereference address in 1. gives
   compute element `j`'s address in row `i`
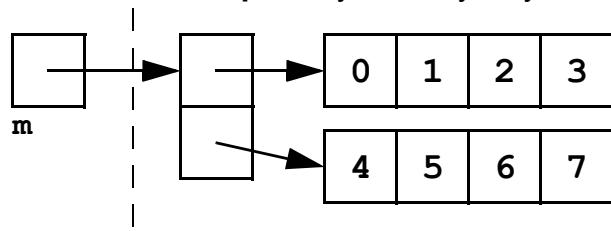   dereference the address in 3. to access element at row `i` column `j`

❋ *m[0][0]*

2D Arrays on the Stack
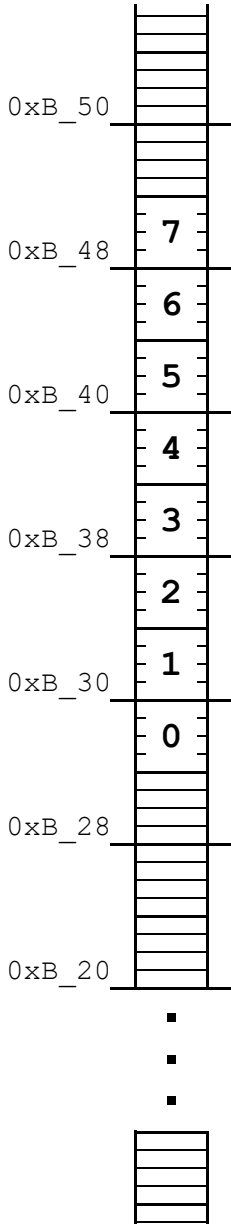
Stack Allocated 2D Arrays in C

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

❋ *2D arrays allocated on the stack*

Stack	
0x_50	
0x_48	7
	6
0x_40	5
	4
0x_38	3
	2
0x_30	1
	0
0x_28	

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`?

   type?
   scale factor?

2. `*m`?   `*(m+i)`?

   type?
   scale factor?

3. `m[0]`? `m[i]`?

4. `m`?

   type?
   scale factor?

For 2D STACK Arrays ONLY

❋ *m and *m are*

❋ *m[i][j]*

2D Arrays: Stack vs. Heap

Stack: row-major order layout

m

0	1	2	3
4	5	6	7

Heap: array-of-arrays layout

m

0	1	2	3

| 4 | 5 | 6 | 7 |

Stack

```
0xB_50
         7
0xB_48
         6
         5
0xB_40
         4
         3
0xB_38
         2
         1
0xB_30
         0
0xB_28
0xB_20
```

Stack

```
0xB_F8   0x0_44
```

Heap

```
         7
         6
0x0_58
         5
         4
0x0_50
0x0_48   0x0_50
         0x0_30
0x0_40
         3
         2
0x0_38
         1
         0
0x0_30
```
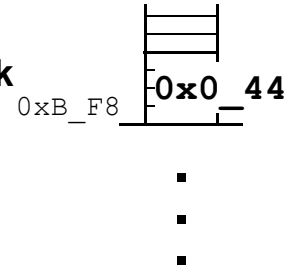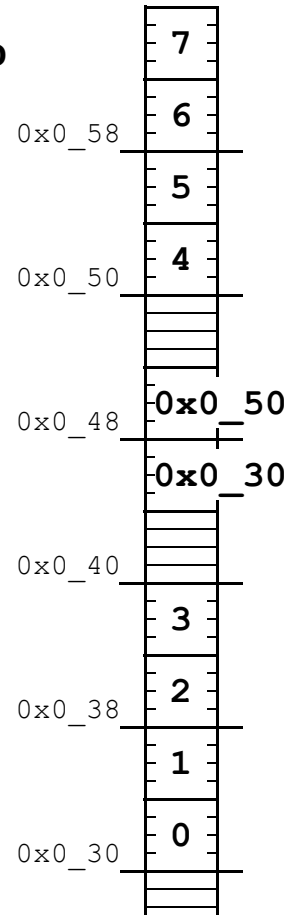
Array Caveats

❋ *Arrays have no bounds checking!*
```
int a[5];
for (int i = 0; i < 11; i++)
   a[i] = 0;
```
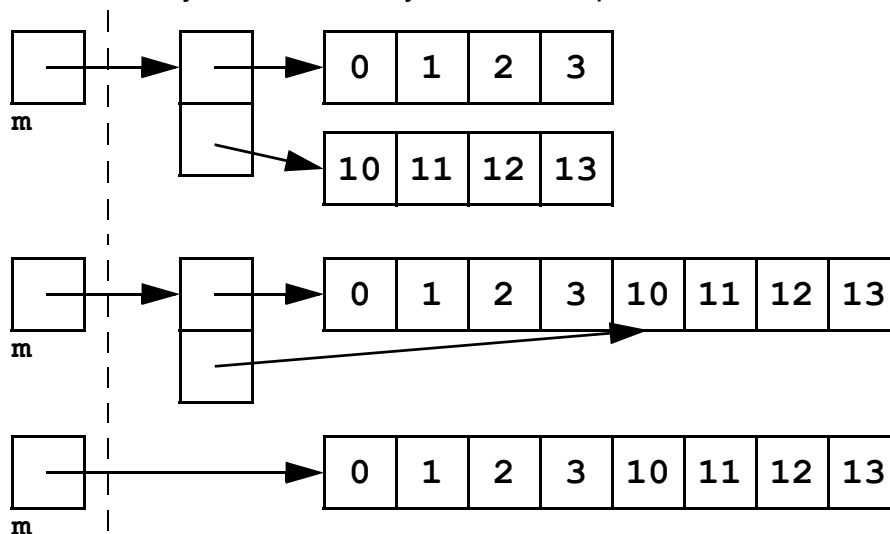
❋ *Arrays cannot be return types!*
```
int[] makeIntArray(int size) {
   return malloc(sizeof(int) * size);
}
```

❋ *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_

    ◆

    ◆

    ◆

    ◆

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;


   } Pokemon;
```
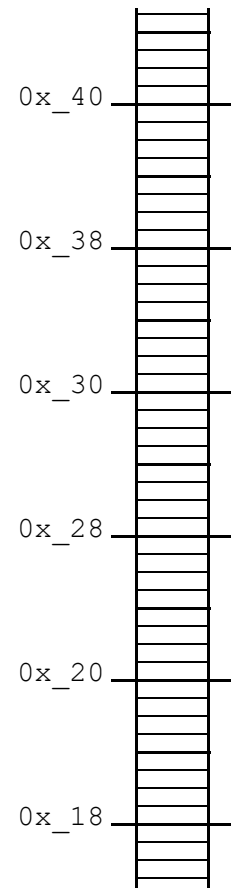
❋ *Structures can contain*

```
                                                          0x_40

                                                          0x_38



                                                          0x_30
```

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

Array of Structures

```
                                                          0x_28
```

❋ *Arrays can have*

```
                                                          0x_20
```

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

```
                                                          0x_18
```

→ 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) {
```

❊ *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

- ◆
- ◆
- ◆
- ◆

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 )
```