

CS 354 - Machine Organization & Programming

Tuesday Feb 7 and Thursday Feb 9, 2023

Project p2A: Due on or before Friday, February 17th

Project p2B: Due on or before Friday, February 24th

Homework hw1 DUE: Monday, February 13th must first mark hw policies

Homework hw2 DUE: Monday February 20st must first mark hw policies

Last Week

Practice Pointers (from L02) Recall 1D Arrays 1D Arrays and Pointers Passing Addresses	1D Arrays on the Heap Pointer Caveats Meet C Strings Meet <code>string.h</code>
---	--

This Week

Tuesday	Thursday
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
<p>Read before next Week</p> <p>K&R Ch. 7.1: Standard I/O</p> <p>K&R Ch. 7.2: Formatted Output - Printf</p> <p>K&R Ch. 7.4: Formatted Input - Scanf</p> <p>K&R Ch. 7.5: File Access</p> <p>Read before next week Thursday</p> <p><u>B&O</u> 9.1 Physical and Virtual Addressing</p> <p><u>B&O</u> 9.2 Address Spaces</p> <p><u>B&O</u> 9.9 Dynamic Memory Allocation</p> <p><u>B&O</u> 9.9.1 The malloc and free Functions</p> <p>Do: Work on project p2A / Start project p2B, and finish homework hw1</p>	

Command Line Arguments

What? Command line arguments are

program arguments:

```
$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:

argv:

→ 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:

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 & 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]`?

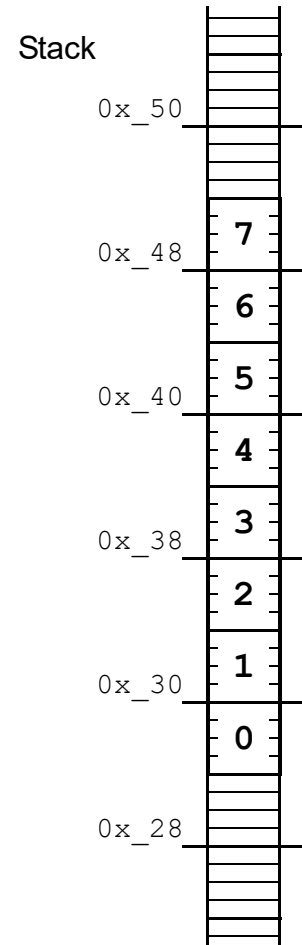
4. `m`?

type?
scale factor?

For 2D STACK Arrays ONLY

✳ *`m` and `*m` are*

✳ *`m[i][j]`*



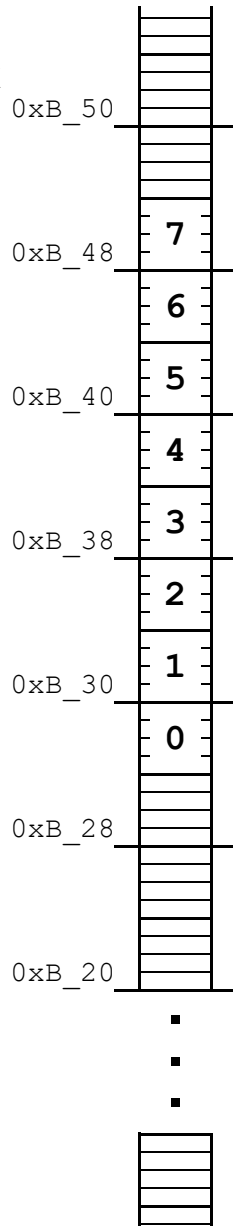
2D Arrays: Stack vs. Heap

Stack: row-major order layout

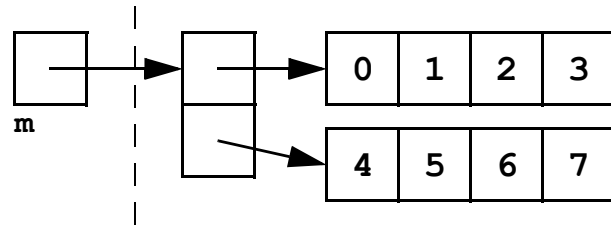
m

0	1	2	3
4	5	6	7

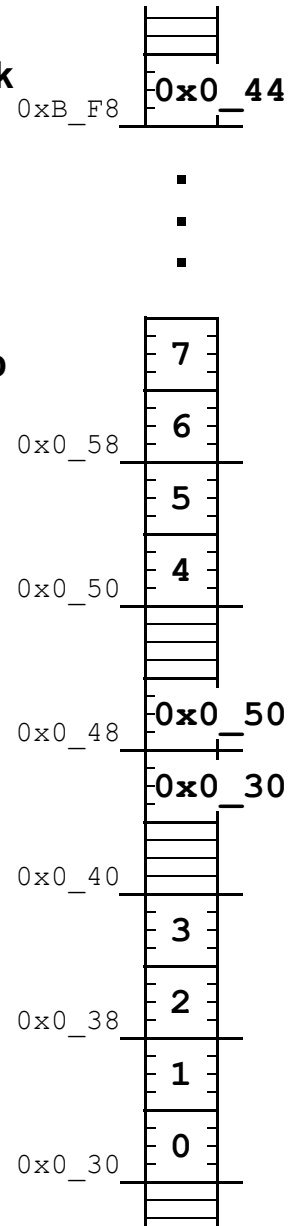
Stack



Heap: array-of-arrays layout



Stack



Heap

Array Caveats

* *Arrays have no bounds checking!*

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

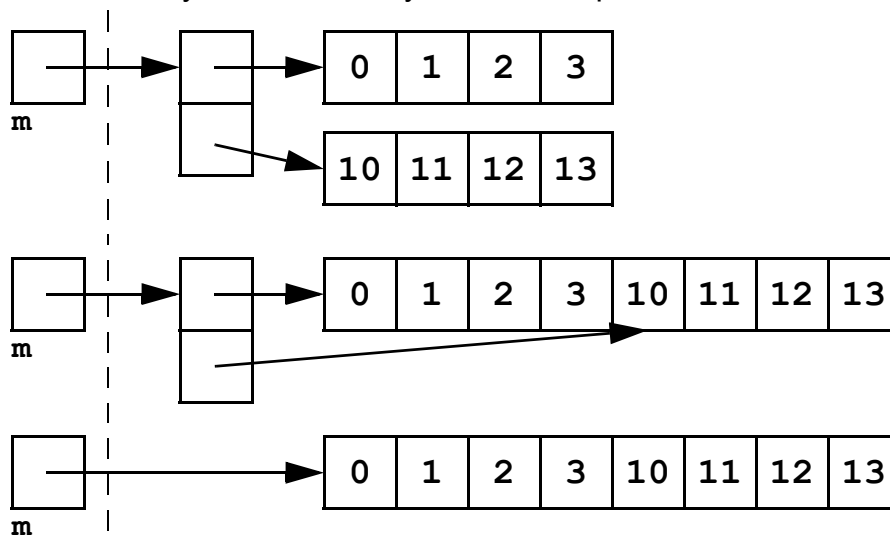
* *Arrays cannot be return types!*

```
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;

    Date  caught;
} Pokemon;
```

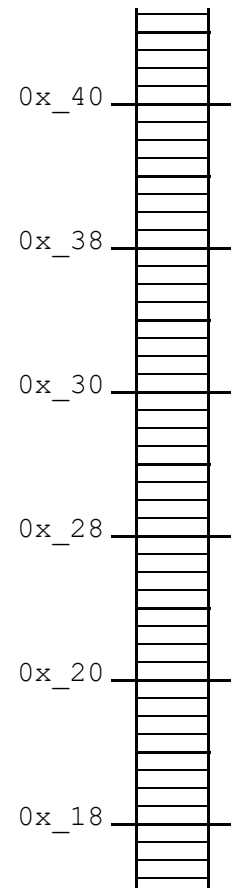
* *Structures can contain*

→ 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) {
```

✱ *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);
}
```