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

CLAs

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

cmd prog args

Why?

enables info to be passed to prog when it begins

How?

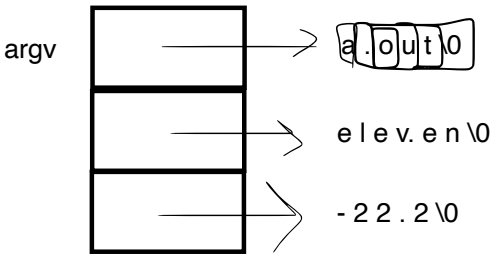
char **argv array of array of char

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

argc: argument count, # of CLA

argv: argument vector, an array of command line arguments

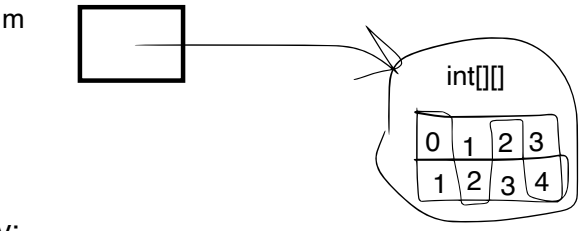
→ Assume the program above is run with the command "`$a.out eleven -22.2`" Draw the memory diagram for argv. argv is on stack



➢ Now show what is output by the program:

output:
a.out
eleven
-22.2

Recall 2D Arrays



2D Arrays in Java

```
                          row  col
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");
}
```

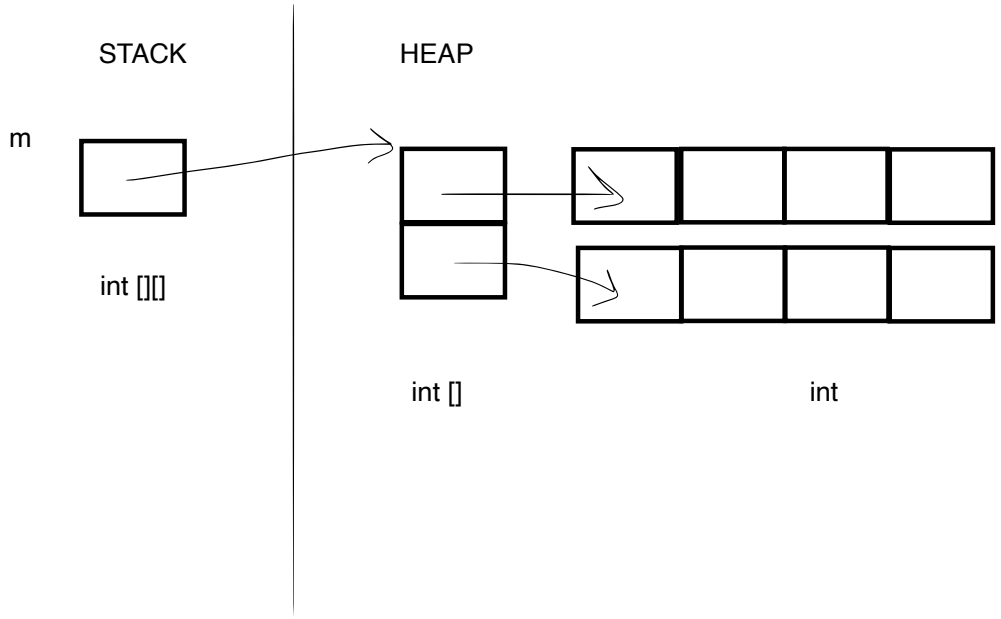
Output:
0 1 2 3
1 2 3 4

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

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

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. integer pointer: int *

so it is int **m; -> right o left: m is a pointer to a pointer to an int

→ **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) …
```

→ **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).

```
for (int i = 0; i < 2; i ++) {
  *(m+i) = 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;
```

save as per last part of java portion

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

```
free(*(m+0))
free(*(m+1))
   free(m)
m = NULL; (gets rid of addr and so no one calls it after)
```

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

❋ *`m[i][j]`* equivalent to *(*(m + i) + j)

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

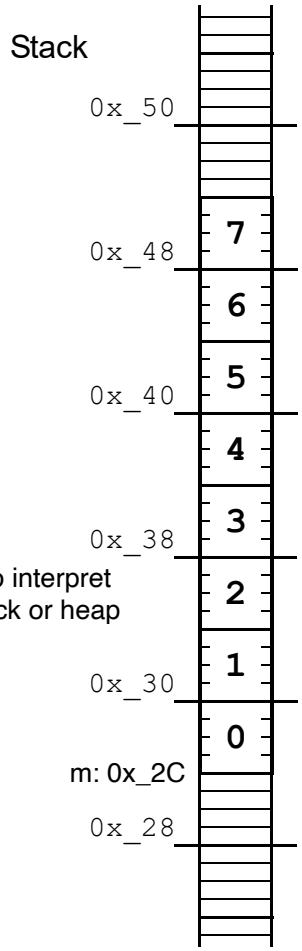
❋ *`m[0][0]`* **m

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 initialize list



❋ *2D arrays allocated on the stack* are laid out in row-major order as a single contiguous block of memory with one row after the other

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`? equivalent to *(*(m)) equivalent to m[0][0] compiler knows how to interpret depending if its on stack or heap

   type? int
   scale factor? none

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

   type? int *
   scale factor? how to skip to row i

   HEAP: sizeof(int *) eg 4 bytes

   STACK: sizeof(int) * (# of COLS) , eg 16 bytes in this case

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

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

   type? int **
   scale factor? to skip to address of "next" element
   STACK: sizeof(element)
   HEAP:

   doesnt really make sense to get to next 2d array element

For 2D STACK Arrays ONLY

❋ *`m` and `*m` are* same address but not same type

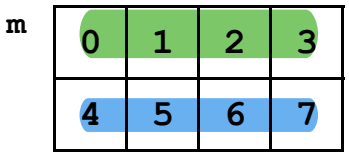
❋ *`m[i][j]`* equivalent to *(*(m + i) + j) equivalent to *(*m + COLS * i + j)

stack & heap use this syntax

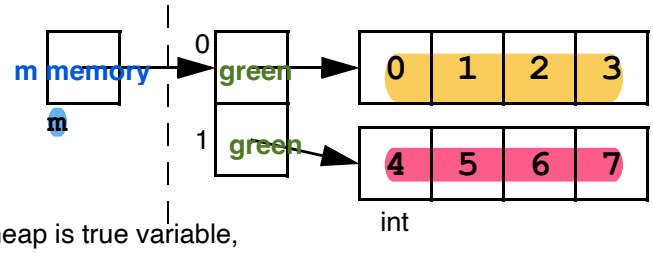
STACK only syntax

2D Arrays: Stack vs. Heap

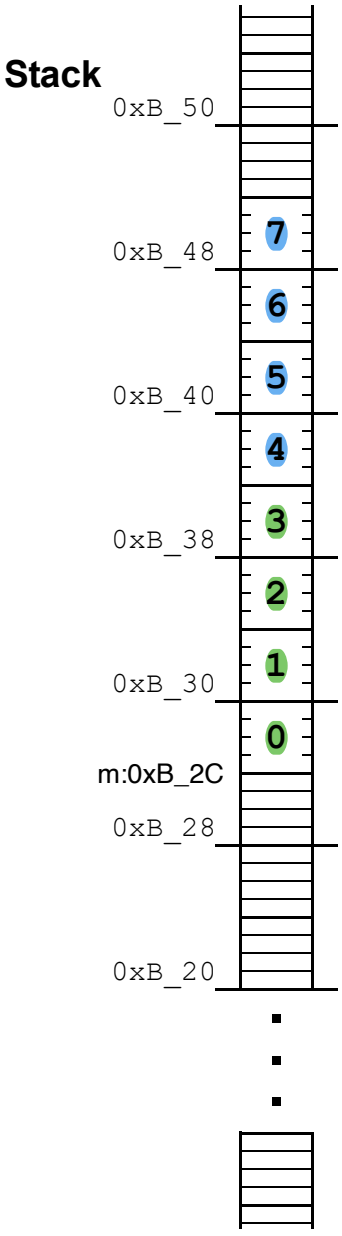
Stack: row-major order layout

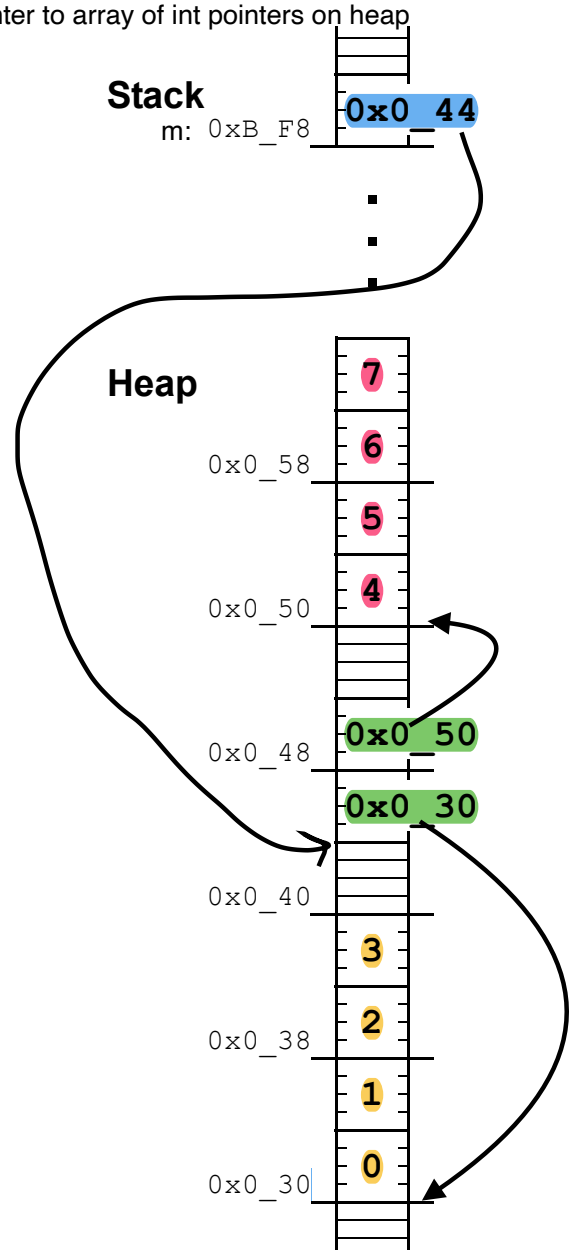


Heap: array-of-arrays layout



m in heap is true variable,
is a pointer to array of int pointers on heap





m[i][j] equivalent to *(*(m+i)+j)

SAA:*(*m + COLS*i + j)

m[i][j] equivalent to *(*(m+i)+j)

Array Caveats

- *Arrays have no bounds checking!*

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

buffer overflow: don't get any error indicators, dangerous

- *Arrays cannot be return types!*

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

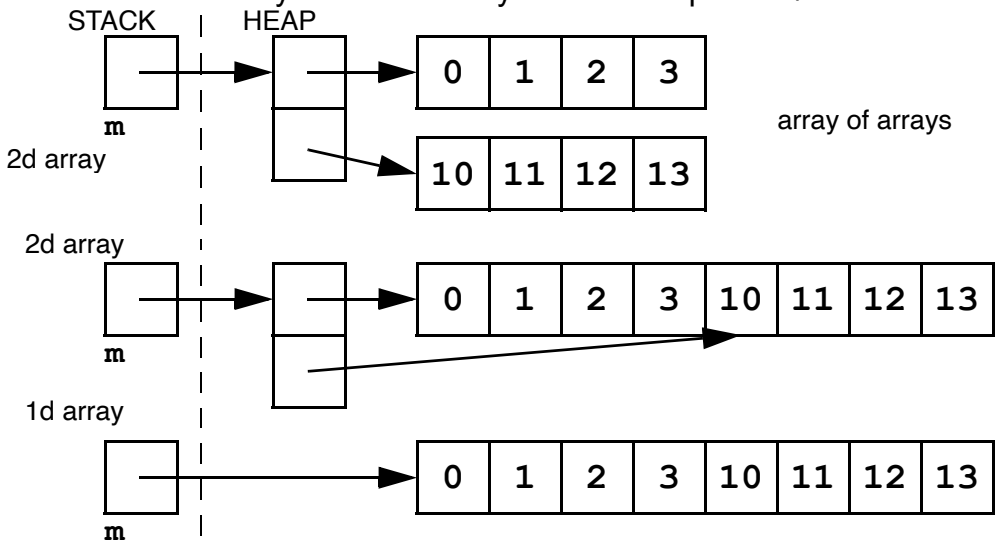
fix is return int * (if on heap)

cannot return address of int on stack: become out of scope, no compiler error, will get indeterminant or runtime error

- *Not all 2D arrays are alike!*

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

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



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

- *Stack allocated arrays require all but their first dimension specified!* argument MUST match type

```
int a[2][4] = {{1,2,3,4},{5,6,7,8}};  can define as int[][4]
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)   ok
void printIntArray(int a[8][4],int rows,int cols)   ok
void printIntArray(int a[][4], int rows,int cols)   ok
void printIntArray(int a[4][8],int rows,int cols)   not ok, type does not match
void printIntArray(int a[][],  int rows,int cols)   not ok, no match, compiler error immediately
void printIntArray(int (*a)[4],int rows,int cols)   ok
void printIntArray(int **a,    int rows,int cols)   not ok
```

init type only has 4 cols
compiler error
when try match type

→ Why is all but the first dimension needed? compiler requires number of columns to find next row
no need next row as it does not do bounds checking

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 contiguous fixed size block of memory

Why?

enables us to organize complex data as a single type

How? Definition 2 ways to define

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

local

usually used locally in same file

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

global

if want to use everywhere, tell compiler to save for later, might use elsewhere

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

```
struct Date {
  int day;
  int month;
  int year;
};
```

```
typedef struct {
  int day;
  int month;
  int year;
} Date;
```

How? Declaration

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

```
struct Date today;
```

type

```
Date today = {8, 2, 2024};
```

```
today.day = 8;
today.month = 2;
today.year = 2024;
```

vs Java, doesnt have constructors, or methods

dot operator: does member selection

- *A structure's data members* are uninitialized by default
- *A structure's identifier used as a source operand* reads the entire struct, if pass struct to another, entire struct gets copied
- *A structure's identifier used as a destination operand* writes entire struct

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

allocated new memory for tomorrow

// is OKAY, copies each member of today to tomorrow

but is slow, usually wants one instance of struct.

Nesting in Structures and Array of Structures

Nesting in Structures

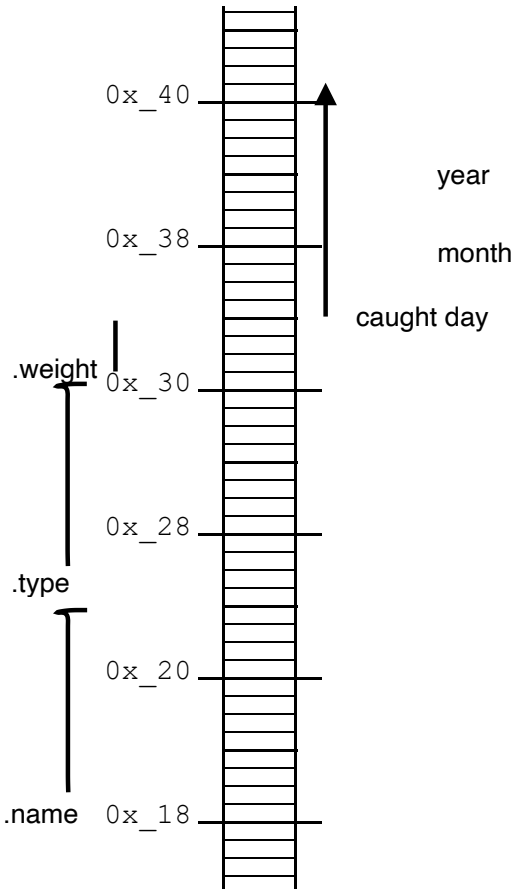
→ Add a `Date` struct, named `caught`, to the structure code below.

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

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

} Pokemon;
```

12 characters = 12 bytes
12 characters = 12 bytes
1 float.= 4 bytes
3 ints in Date so 12 bytes

❋ *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* structs for elements

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

```
Pokemon pokedex[2] = {
  {"Abra", "psychic", 43.0, {1, 21, 2024}},
  {"Oddish", "grass", 22.9, {9, 22, 2023}}
};
```

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

```
pokedex[1].weight = 22.2;
```

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

```
pokedex[0].caught.month = 11;
```

Passing Structures

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

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

%2i prints 2 chars

❋ *Structures are passed-by-value to a function,* which copies the entire struct

SLOW

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) {

        for (int i = 0; i < size; i ++) {
          printPm(dex[i]);
        }


}
```

❋ *Recall: Arrays are passed-by-value to a function,* but only starting address is passed

FAST

Pointers to Structures

Why? Using pointers to structures

- to avoid copy overhead of pass-by-value
- allows function to change struct data members
- enables heap allocation of structs
- enables linked structs

How?

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

```
Pokemon *pmPtr;
pmPtr = malloc(sizeof(Pokemon));
```

allocated 4 bytes, as it is an address, address always 4 bytes since 32 bit machine

→ Assign a weight to the `Pokemon`.

```
(*pmPtr).weight = 4.2;
```

points-to operator: dereferences first, then selects data member ->
points-to operator

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

```
strcpy(pmPtr->name, "Abra");
```

have to use strcpy, as cannot assign strings this way, have to copy one character at a time, using loop or strcpy

→ Assign a caught date to the `Pokemon`.

```
pmPtr -> caught.mon = 2;
pmPtr -> caught.day = 13;
pmPtr -> caught.year = 2024;
```

→ Deallocate the `Pokemon`'s memory.

```
free(pmPtr);
```

→ 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 )
```

pass pointer instead of whole pokemon struct

STACK