CS 354 - Machine Organization & Programming
Tuesday Sept 19, and Thursday Sept 21, 2023

Project p2A: Due on or before Friday, Sept 29

Project p2B: Due on or before Friday, Oct 7th (due after E1, but should be written before E1)

Homework hw1 DUE: Monday Sept 25, must first mark hw policies page

Homework hw2 DUE: Monday Oct 2nd, 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

<table>
<tr><th>Tuesday</th><th>Thursday</th></tr>
<tr><td>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</td><td>Array Caveats
Meet Structures
Nesting in Structures and Arrays of Structures
Passing Structures
Pointers to Structures</td></tr>
<tr><td colspan="2">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

<b>Do:</b> Work on project p2A / Start project p2B, and finish homework hw1 (arrays and pointers)</td></tr>
</table>

Command Line Arguments

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

check_board board.txt
program arg

2 CLAS

program arguments: arguments that follow command

CLAs

program arg

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

command

output name

count as 2CLA

(7) CLAS

Why?

enables info to be pushed 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 char *

argc: argument count , # CLA

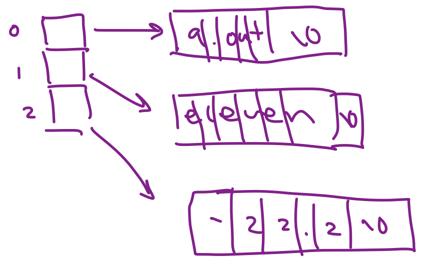
argv: argument vector

→ 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

Mem diagram

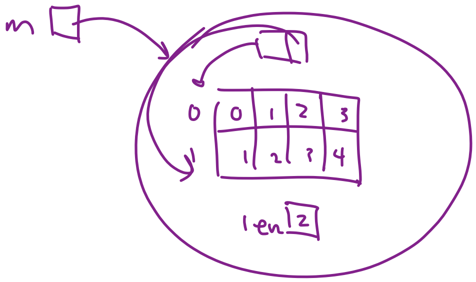


Recall 2D Arrays

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;
```

*(m+i) ??

➢ 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");
}
```

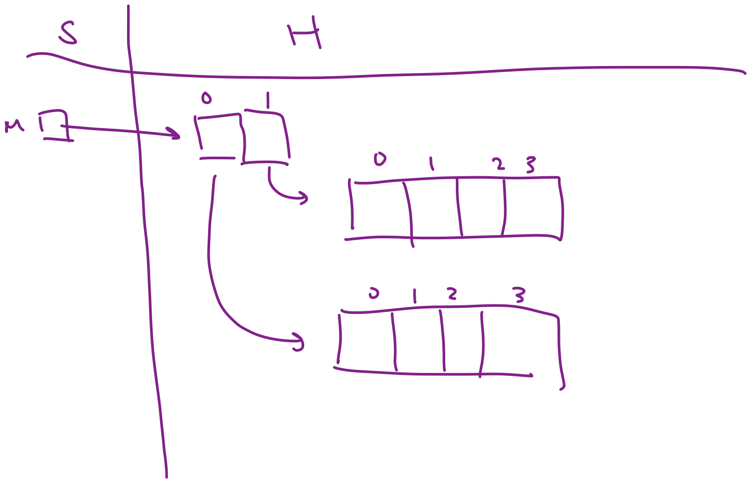
→ What memory segment does Java use to allocate 2D arrays?

heap

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

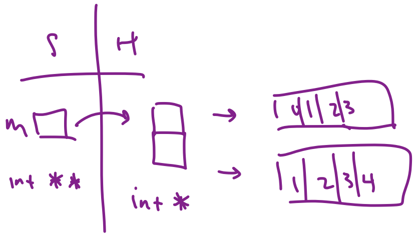
1D array of pointers to 1D arrays

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



2D Arrays on the Heap (C)

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= maloc (size of (int) · 2));

if (m == NULL) { print , exit() }

→ **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).

if (m[0]==NULL) * (m + 0) = malloc (size of (int) · 4));

if (m[1] ==NULL) m[1] = malloc (" ");

➤ 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.

free(m[0]

free (* (m+)) ;

free (m);

❈ *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]`?

↑ 1st D ↑ 2nd D

a.) `*(m[i]+j)` ok
b.) `(*(m+i))[j]` ok
c.) `*(*(m+i)+j)` ok , p 2 A & p 2 B

❈ *m[i][j]* ≡ *(*(m+i) + j)

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

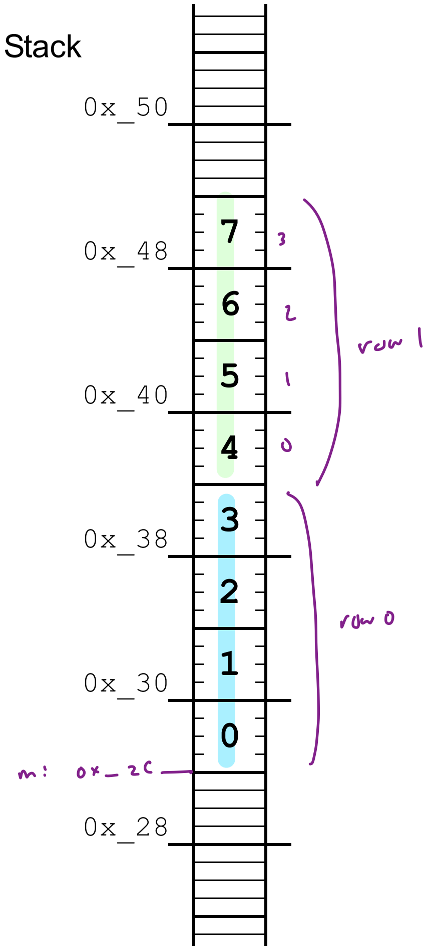
❈ *m[0][0]* ≡ *(*(m+0) +0) ≡ *(*(m)) ≡ **m

2D Arrays on the Stack (C)

Stack Allocated 2D Arrays in C

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

dimensions in declaration - SAA   row 0   row 1



❋ *2D arrays allocated on the stack* are laid out in row major order as a continous block

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+0) + 0) ≡ m[0][0]

   type? int
   scale factor? none

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

   type? int *
   scale factor? skip to row i?
   STACK = 16 bytes = 4 elements * 4 bytes/element
   HEAP = 4 bytes = 4 bytes / element = next row int *

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

4. `m`? STACK: addr to start (1st elt) of 2D SAA
   HEAP: addr of 1D array of int *

   type? int **
   scale factor? to skip to addr of next (row)?
   STACK: elt size * # of columns
   HEAP: size of (int *)

For 2D STACK Arrays ONLY

❋ *m* and **m* are

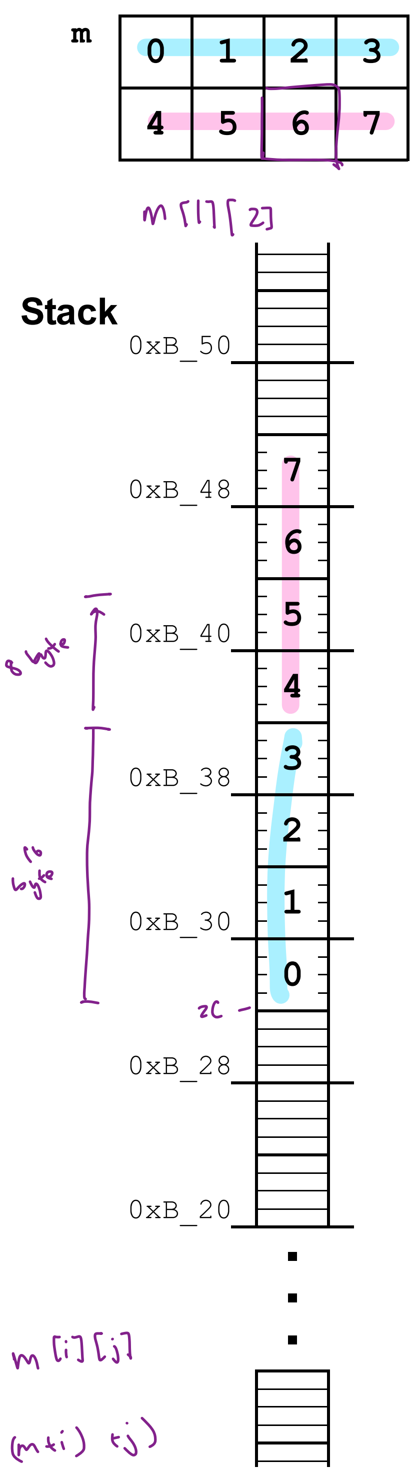
❋ *m[i][j]* ≡ *(*(m+i) +j) ≡

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

multip

2D Arrays: Stack vs. Heap

Stack: row-major order layout

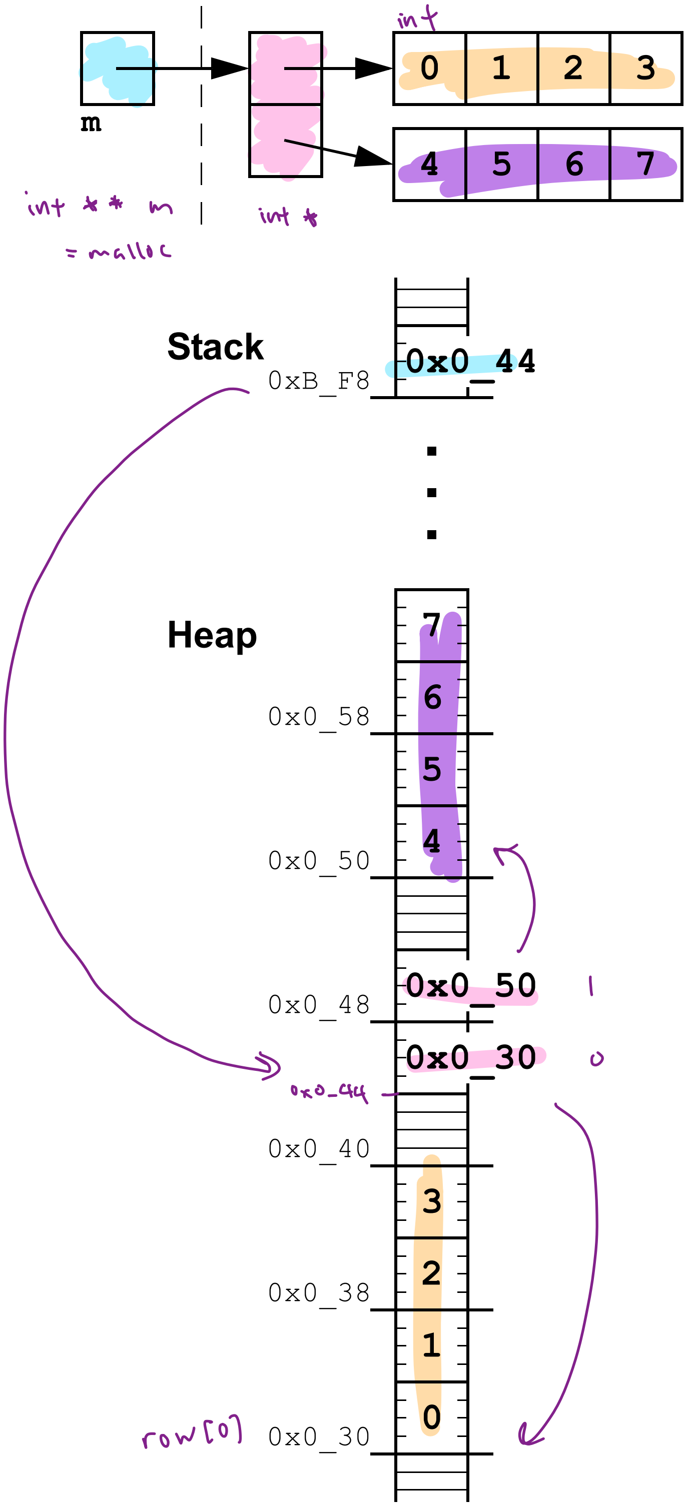


m[i][j]

((m+i) +j)

*(*m+ COL·i +j)

Heap: array-of-arrays layout



m[i][j]

*(*(m +i) +j)

Array Caveats

❋ *Arrays have no bounds checking!*

```
int a[5];
for (int i = 0; i < 11; i++)   // Buffer overflow        // overwrite the stack
   a[i] = 0;                   intermittent error
```

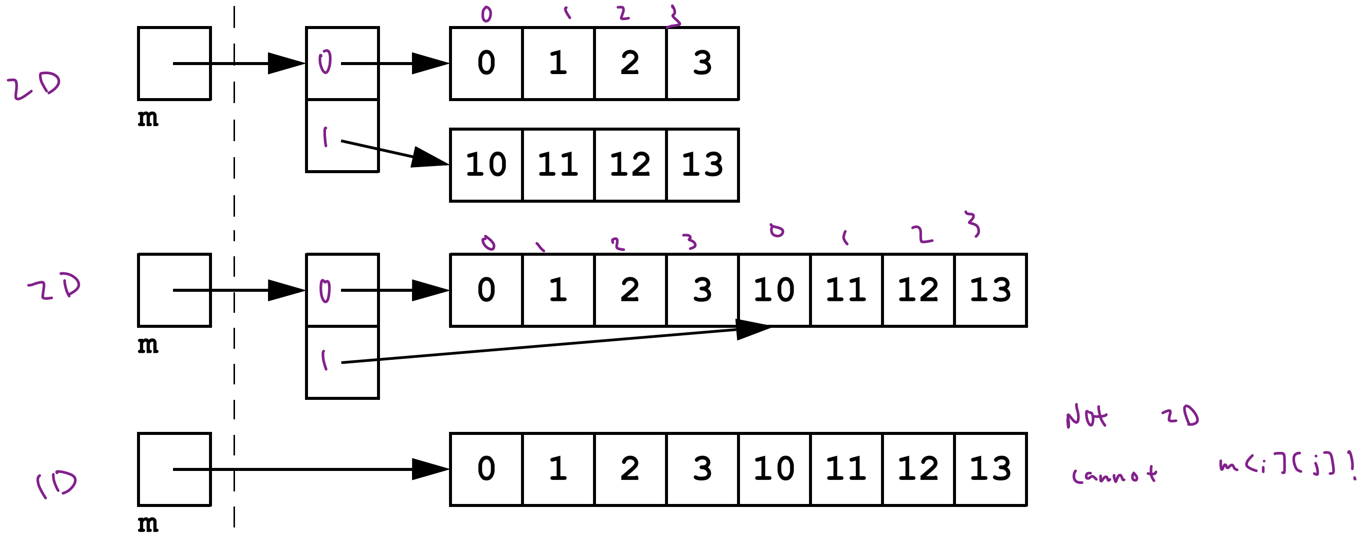
❋ *Arrays cannot be return types!*

```
        int[] makeIntArray(int size) {     // compiler error
use int*   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?

```
   Y void printIntArray(int a[2][4],int rows,int cols)
                                    one match
→  Y void printIntArray(int a[8][4],int rows,int cols)
   Y void printIntArray(int a[][4], int rows,int cols)
                                    none match
→  N void printIntArray(int a[4][8],int rows,int cols) compiler needs # of cols
→  N void printIntArray(int a[][],  int rows,int cols)     "
   Y void printIntArray(int (*a)[4],int rows,int cols)
   N void printIntArray(int **a,    int rows,int cols)
                        a is a stack array!
```

→ Why is all but the first dimension needed? compiler only needs dims to generate code

* (* a + 4*i + j)

Meet Structures in C

What? A *structure*

- user - defined type
- a compound unit of storage w/ data members of diff types
- access using identifier and data member name
- contiguous fixed-size block of memory

Why?

enables organizing complex data as a single entity

How? Definition

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

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

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

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

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

How? Declaration

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

type

```
struct Date today;
    today.month = 9
    today.day = 21
    today.year = 2023
```

type

```
Date today = {9,21,2023};
```

dot operator: does member selection

❈ *A structure's data members* are uninitialized by default

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

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

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

ok

// copies each member of today to tomorrow

Nesting in Structures and Array of Structures

Nesting in Structures

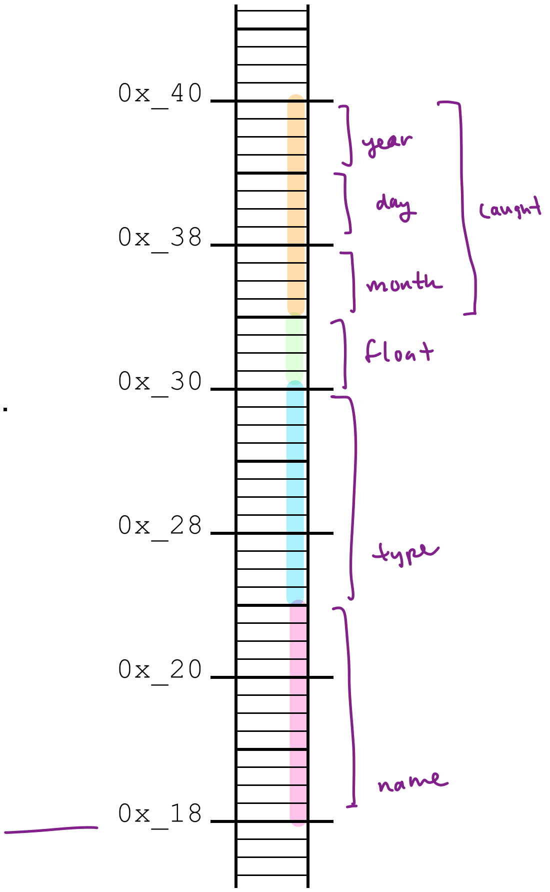
→ Add a `Date` struct, named `caught`, to the structure code below.

```
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 as deeply as you wish

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

at address 0x _ 18



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, 2020}},
   { "oddish", "Grass", 33.2, { 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.

Passing Structures

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

```
void printDate (Date date) {   // mm/dd/yyyy
      printf(" %2i / %02i / %i \n", date.month, date.day, date.year);
                      ↑
                 proceeding 0
}
```

✵ *Structures are passed-by-value to a function,* which copies 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);
   ...
```

entire struct copied

→ 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]);
                        ↖ addr!
                          not Pkmn!
}
```

✵ *Recall: Arrays are passed-by-value to a function,* but only starting addrs

Fast

Pointers to Structures

Why? Using pointers to structures

- avoid copying overhead of pass-by-value
- allows func to change struct's data members
- allows heap allocated structs
- enables linked structs

How?

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

```
Pokemon *pmptr;        pmptr = malloc( sizeof(Pokemon));
                       if (pmptr == NULL) ...
```

→ Assign a weight to the `Pokemon`.

```
(*pmptr).weight = 43;
```

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

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

```
strcpy( pmptr -> name, "Abra")
```

→ Assign a caught date to the `Pokemon`.

```
pmptr -> caught. month = 9;
               .day = 22;
               .year = 2023;
```

→ Deallocate the `Pokemon`'s memory.

```
free(pmptr)
```

when stack allocated mem is freed @ return

→ 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 )
```

stack