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

exam conflicts due this week

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 `string.h`

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

`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

Command Line Arguments (CLAs)

What? *Command line arguments* are white space separated list of input entered after command on command line

program arguments: follow command of program name

CLAs

program arguments

shell prompt

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

command

Why? enables info to be passed to program when it begins

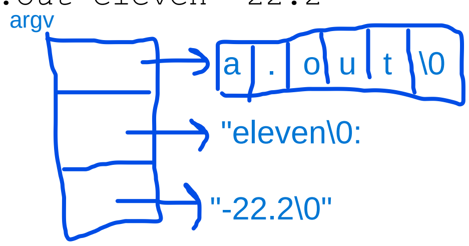
How? array of ptr to char, array of C strings
char **argv (in p2b)

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

argc: argument count (# of CLAs)

argv: argument vector (array of character pointers)

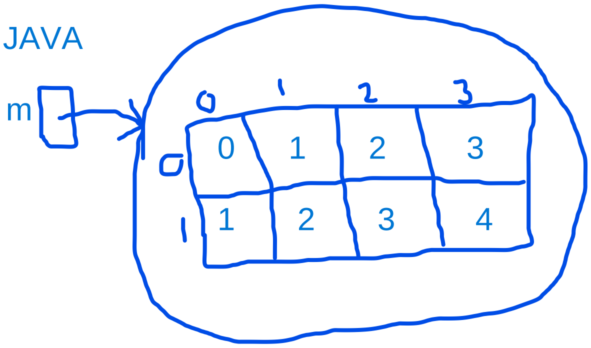
→ Assume the program above is run with the command "`$a.out eleven -22.2`"
Draw the memory diagram for argv.



➢ Now show what is output by the program:

output
a.out
eleven
-22.2

Recall 2D Arrays IN JAVA



2D Arrays in Java

rows columns

```
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;
```

row index column index

➢ 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");
}
```

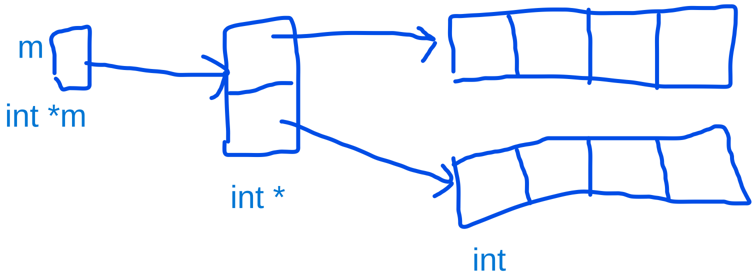
output

```
0123
1234
```

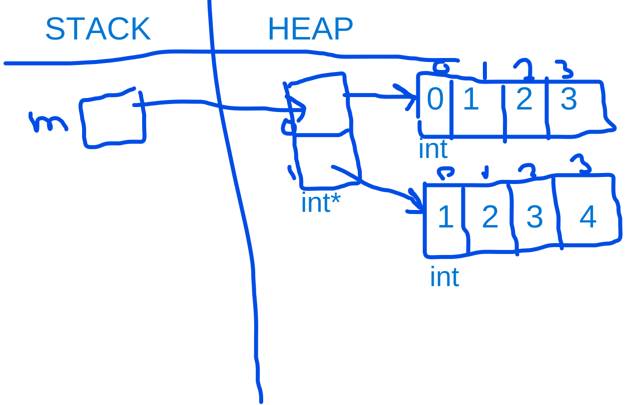
→ What memory segment does Java use to allocate 2D arrays? HEAP

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

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



2D Arrays on the Heap IN 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 = malloc( 2 * sizeof(int*));
if (m == NULL) { print ("not enough memory\n"); exit(i);}
```

ALWAYS NEED IN CODE

→ **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 r = 0; r < 2; r++) {
        *(m + r) =malloc(4 * sizeof(int));
        if (*(m+r) == NULL) { print ("not enough memory\n"); exit(i);}
}
```

➢ 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[1]);
free(m)
```

could be in loop

ALSO ADD:
m[0] = NULL;
m[1] = NULL;
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)` okay
b.) `(*(m+i))[j]` okay (don't do it though)
P2A & P2B c.) `*(*(m+i)+j)` Yes, okay

❋ *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+i
*( )
+j
*( )

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

≡ *(*(m+0))

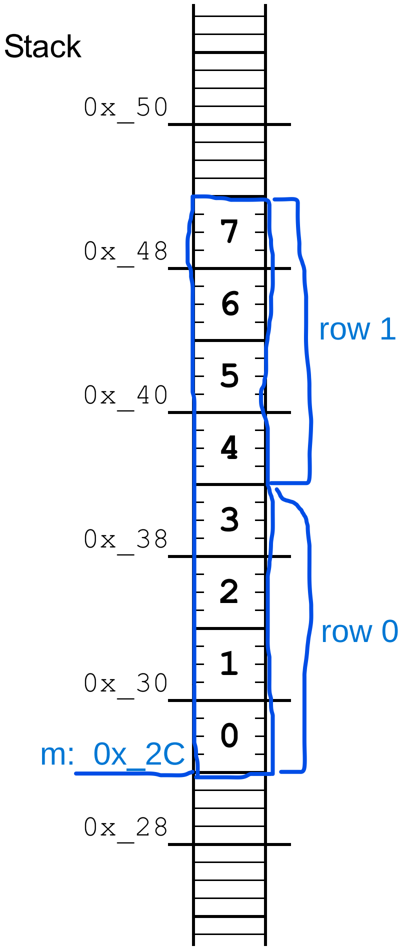
2D Arrays on the Stack

Stack Allocated 2D Arrays in C

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

row 0 row 1

Stack Allocated Array init 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 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[0][0]

   type? int
   scale factor? NONE

2. `*m`? `*(m+i)`? address of start of row 0, row i

   type? int *
   scale factor? skip to next item, next row
   STACK: 4 int with 16 for size of row (4 byte * 4 int)
   HEAP: 4 sizeof (int*)

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

4. `m`? STACK: address of start of 2D array
   HEAP: address of ID Aray of Arrays

   type? int **
   scale factor? to skip to address of next row
   STACK: 16 = 4 byte * 4 ints
   HEAP: 4 = sizeof(int*)

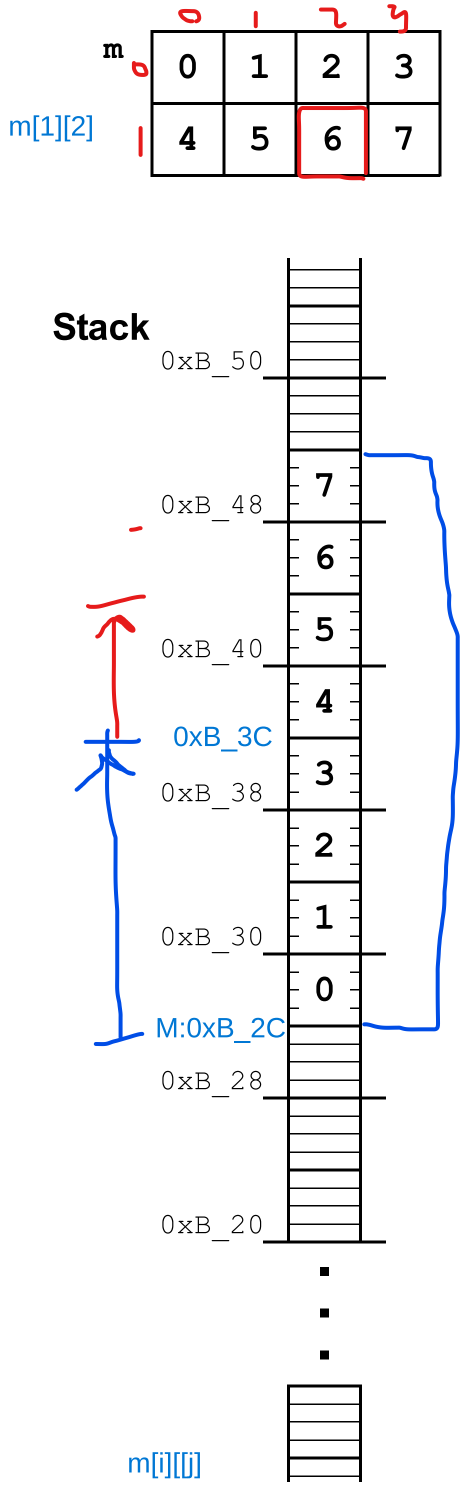
For 2D STACK Arrays ONLY

❈ *m* and **m* are the same address but not the same type

❈ *m[i][j]* = *(*(m+i)+j) = *(*m + columns * i + j) - ONLY WORKS FOR STACK ALLOCATED

2D Arrays: Stack vs. Heap

Stack: row-major order layout



((m+i)+j)

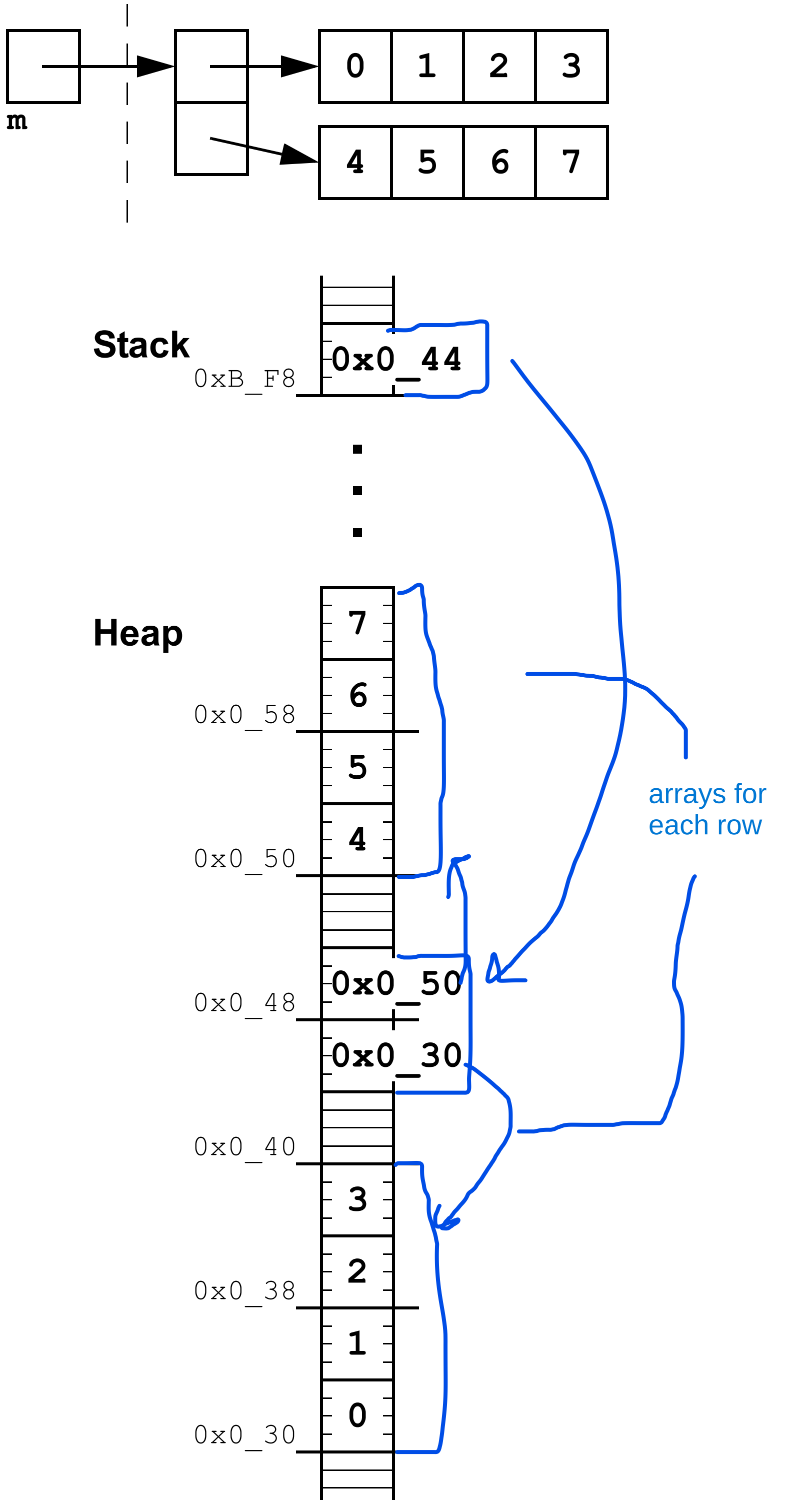
SAA: *(*m + cols*i + j)

*(0xB_2C + 4 * i + 2)

start row 1 col 2

0xB_2C
+ 10
0xB_3C

Heap: array-of-arrays layout



m[i][j] = *(*(m+i)+j)
~~*(*m + columns * i + j)~~ not for heap allocated arrays

Array Caveats

✵ *Arrays have no bounds checking!*

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

✵ *Arrays cannot be return types!* COMPILER ERROR (when using int[] as return type)

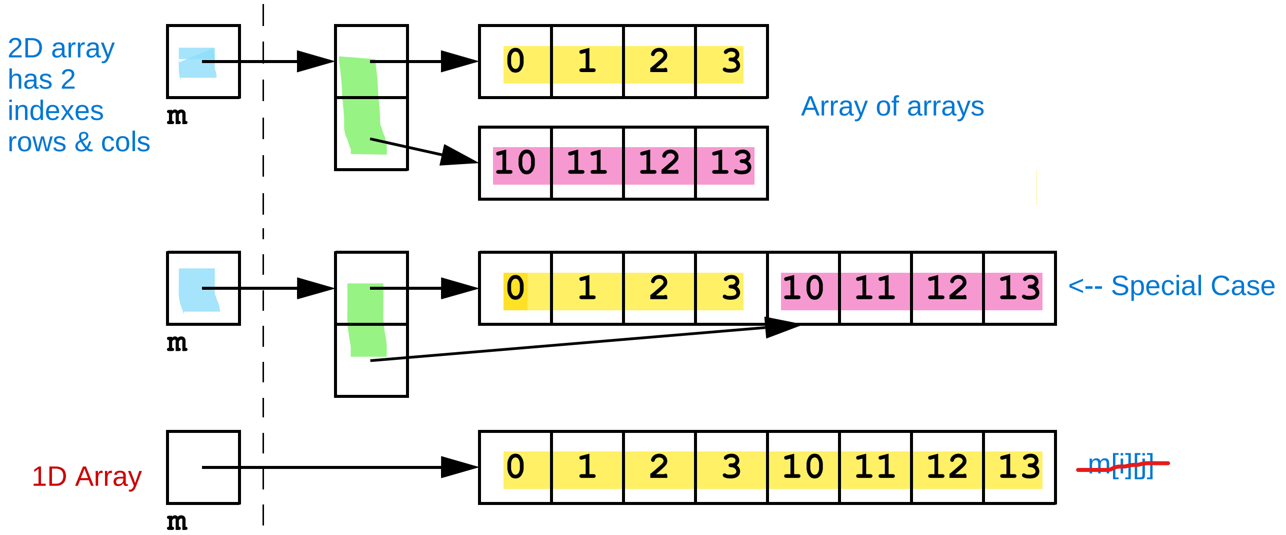
use int* instead

```
~~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? contiguous block in row-major order

→ What is the layout for 2D arrays on the heap? "Array of Arrays"



✵ *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? NO OVERLOADING IN C, can't have multiple functions w/ same name

```
okay void printIntArray(int a[2][4],int rows,int cols)
okay void printIntArray(int a[8][4],int rows,int cols)
okay void printIntArray(int a[][4], int rows,int cols)
     void printIntArray(int a[4]~~[8]~~,int rows,int cols) COMPILER ERROR
     void printIntArray(int a[][],  int rows,int cols) COMPILER ERROR
okay void printIntArray(int (*a)[4],int rows,int cols)
     void printIntArray(int **a,    int rows,int cols) doesn't work for stack allocated array
```

→ Why is all but the first dimension needed? compiler only needs number of columns to find next row

C

Meet Structures

What? A *structure*

- user-defined type
- a compound unit of storage with data members of different types
- access using identifier and data member name
- allocated as a contiguous fixed-size block of memory

Why?

enables organizing complex data as a single module

How? Definition

2 name spaces

possible to have same typename in local and global structs in the same file (don't do it though)

local

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

global

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

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

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

- improved readability over arrays
- compiler will help more

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

How? Declaration

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

```
struct Date today;
```

type identifier

```
today.mon = 2;
today.day = 9;
today.year = 2023;
```

initializer list can be used for both

```
Date today = {2, 9, 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 struct

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

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

copies each member of today to tomorrow

Nesting in Structures and Array of Structures

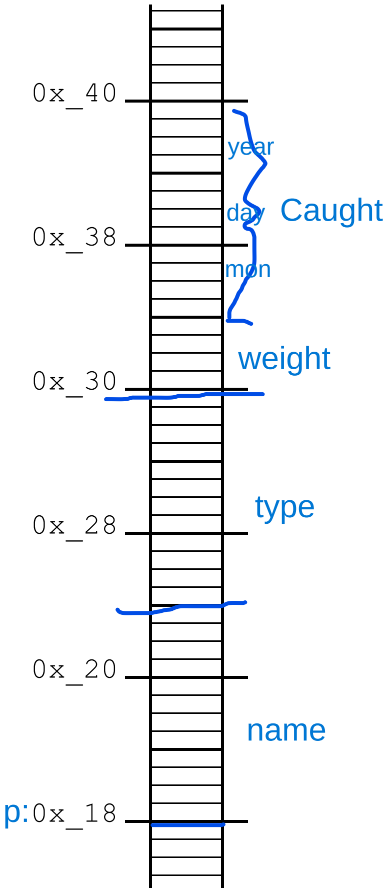
Nesting in Structures

→ 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 and arrays nested as deeply as you wish

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

if it starts 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, {2, 21, 2021}},
   {"Oddish", "grass", 22.9, {9, 22, 2022}}
};
```

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

```
pokedex[1].weight = 22;
```

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

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

Passing Structures

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

```
void printDate (Date date) {  //mon/day/year
    printf("%i/%02i/%i\n", date.mon, date.day, date.year);
                  02 forces 2 digits to be present, precedes with 0's if int is 1 digit
}
```

❈ *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);
   ...
```

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

```
void printDex(Pokemon dex[], int size) {
    for(int i = 0; i < size; i++)
      print Pm(dex[i]);
    }
```

❈ *Recall: Arrays are passed-by-value to a function,* but only starting address is passed and copied. The array elements are not copied.

FAST

Pointers to Structures

Why? Using pointers to structures

- to avoid copying overhead from pass-by-value for struct
- allows functions to change struct data members
- enable heap allocation of struct
- enables creating linked structures

How?

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

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

→ Assign a weight to the `Pokemon`.

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

. takes precedence over *, make sure to put parentheses around *pmptr

points-to operator: dereference 1st, then member selection

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

~~pmptr -> name = "Alona";~~ //can't copy using assignment

```
strcpy(pmptr -> name, "Alona");
              dest           src
```

→ Assign a caught date to the `Pokemon`.

```
pmptr -> caught.mon = 2;
pmptr -> caught.day = 14;
pmptr -> caught.year = 2023;
```

→ Deallocate the `Pokemon`'s memory.

```
free(pmptr);    //do NOT need to free pmptr's variables individually
pmptr = null;
```

→ 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");                                Date
}
int main(void) {
   Pokemon pm1 = {"Abra","Psychic",30,{1,21,2017}};//STACK
   printPm( &pm1)
```