CS 354 - Machine Organization & Programming
Tuesday, September 24, 2019

Project p2A (3%) DUE: 10 pm, Monday, September 30th

Project p2B (3%) ASSIGNED TOMORROW

Homework hw1 (1.5%) DUE: 10 pm, Friday, September 27th

Homework hw2 (1.5%) (3%) ASSIGNED TOMORROW

Last Time

Meet `string.h`
Recall 2D Arrays
2D Arrays on the Heap
2D Arrays on the Stack
2D Arrays: Stack vs. Heap
Array Caveats

Today

Array Caveats (from last time)
Command-line Arguments
Meet Structures
Nested Structures and Arrays of Structures
Passing Structures
Pointers to Structures

Next Time

Quick Review of I/O
Read:
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
C Abstract Memory Model
Read:
B&O 9.1, 9.2, 9.9 (upto 9.9.1)

Command Line Arguments

(command line).

Cl. Args

What?

Prog args

Consider the Linux command: `$gcc myprog.c -Wall -m32 -std=gnu99 -o myprog`

info entered at command prompt ($).

note: command line args include the program name

Why?

enables info to be passed to a prog before it execute

How?

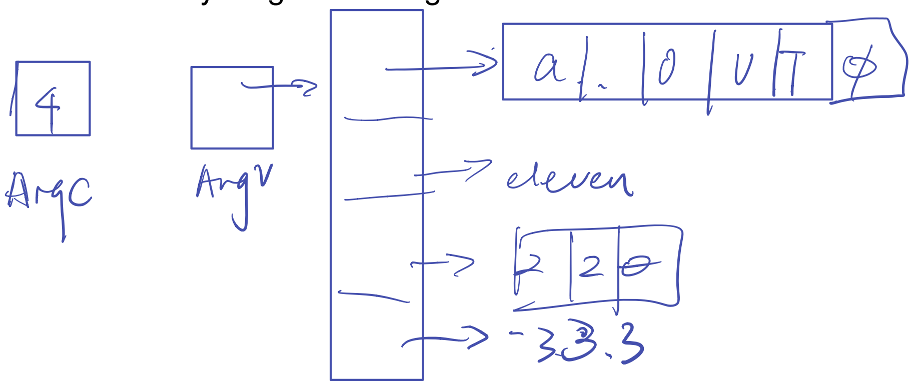
Consider the code:

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

argc: arg count of whitespace seperated ClA's

argv: Arg values, which is an array of char pointers where each points to a string for each ClA.

→ Assume the program above is invoked with "`a.out eleven 22 -33.3`"
Draw the memory diagram for argv.



➢ Now show what is output by the program:

Copyright © 2016-2019 Jim Skrentny

Meet Structures

What? A structure defines: a new data type that is a compositive of related info of any type. called data members

Why? enables organizing into a single module Data that's often of different types.

How?

```
struct <struct-name> {          is a seperate name space
   <data-declarations>;
} <optional-list-of-variables>;      the "struct" name
```

this is required to finish struct Defn

→ Declare a structure representing a date having a integer month, day of month, and year.

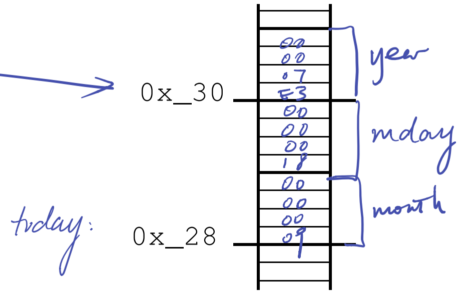
```
struct date {
    int month;
    int mday;
    int year;
};
```

```
typedef struct <name> {
(define a type)       ↑
                   optional
    int month;
    int mday;
    int year;
} Date;
```

→ Create a variable containing today's date.

```
struct date today;
today.month = 9;
today.mday = 24;
today.year = 2019;
```

```
Date today;
```



dot operator (.): does memory access.

Typedef

what: define a new type name in global name space

why: less cluttered

→ Update the code above to use typedef. see green code

✵ *Limit your use of typedef* To things like struct types.

Nested Structures and Array of Structures

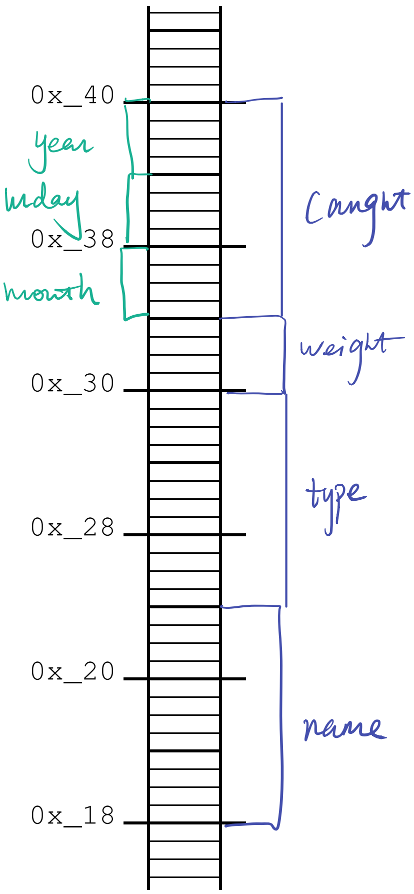
Nested Structures

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

```
typedef struct { ... } Date; //from previous page

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

} Pokemon;
```



✵ *Structures can contain* other structures 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* structures for their element

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

```
pokemon pokedex [ 2 ] =
  { { "Abra", "Psychic", 43.0, { 1, 21, 2017 } },
    { "addish", "Grass", 11.9, { 2, 24, 2018 } } };
```

→ 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) {
  printf("%i/%i /%i", date.month, dath.mday, day. year);
}
```

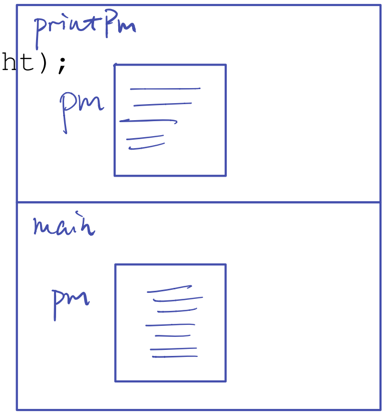
✵ *Structures are passed-by-value to a function,* but the entire structhre is copied.

Consider the additional code:

```
//assume code for Date, Pokemon, printDate is included here

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 `pokedex`.

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

✵ *Arrays are passed-by-value to a function,* but only their address is copy. The array's elements are not copied.

Copyright © 2016-2019 Jim Skrentny

Pointers to Structures

Why? Pointers to structures

- enables heap allocation of structs
- avoid copying overload of structs.
- allows functions to change structs args passed to them.
- enables creating linked data structs.

How?

→ Declare a pointer to a `Pokemon`.

Pokemon *p;

→ Dynamically allocate space for a `Pokemon`.

p = malloc(sizeof(Pokemon));

→ Assign a weight to the `Pokemon`.

(*p).weight = 77; needed to force dereference over member access

p -> weight = 77; preferred.

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

~~p -> name = "Abra";~~ (can't assign).

strcopy (p -> name, "Abra");

- - - (p -> type, . ——.

→ Assign a caught date to the `Pokemon`.

p -> caught. month = 9; ~~= {9, 20, 2016};~~

p -> caught . day = 20;

p -> caught . year = 2016;

→ Deallocate the `Pokemon`'s memory. whether need to free individually: one piece of mem or not.

free(p); P=NULL

→ Update `printPm` 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);
                ↑ copy
```