outlineL06-w3TR-student (1)

CS 354 - Machine Organization & Programming
Tuesday Sept 20 and Thursday, Sept 22, 2022

Project p2A: Due on or before Friday, Sept 30th,
Project p2B: Due on or before Friday, Oct 7th
Homework hw1 DUE: Monday Sept 26th, must first mark hw policies
Homework hw2 DUE: Monday Oct 3rd, 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 `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

 CS 354 (F22): L6 - 1

CS 354 - Machine Organization & Programming
Tuesday Sept 20 and Thursday, Sept 22, 2022

Project p2A: Due on or before Friday, Sept 30th,
Project p2B: Due on or before Friday, Oct 7th
Homework hw1 DUE: Monday Sept 26th, must first mark hw policies
Homework hw2 DUE: Monday Oct 3rd, 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 `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

Copyright © 2016-2022 Jim Skrentny **CS 354 (F22): L6 - 1**

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:

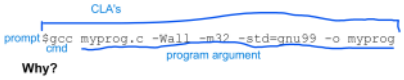
→ Assume the program above is run with the command "`$a.out eleven -22.2`"
Draw the memory diagram for argv.

➢ Now show what is output by the program:

Command Line Arguments

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

program arguments: follow command or program name



Why?

enables us to pass info to program at start

How?

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

argc: argument count

argv: argument vector (list or array)

→ 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?

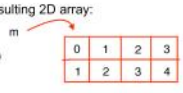
Recall 2D Arrays

2D Arrays in Java

```
int[][] m = new int[2][4];
```
Always Rows x Columns

→ 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;
```



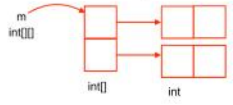
➢ What is output by this code fragment?

```
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? 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?



No null terminating characters since this isn't a string/array of characters.

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)`

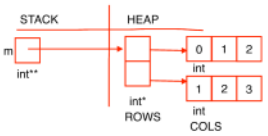
❋ *`m[i][j]`*

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 Heap

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 = (int**)malloc(sizeof(int*) * 2); //This creates the row array for the 2D array`

→ **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) = (int*)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;
```

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

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

❋ *Avoid memory leaks; free the components of your heap 2D array*
In reverse order of allocation (start with int* -> int**)

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]`* = `*(*(m + i) + j)`

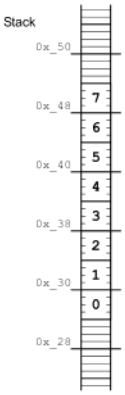
compute row `i`'s address `m + i`
dereference address in 1. gives `*(m + 1)` is the row.
compute element `j`'s address in row `i` + j portion as that will choose the value in the column.
dereference the address in 3. to access element at row `i` column `j` `*(*(m + i) + j)`

❋ *`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}};
```



❋ *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

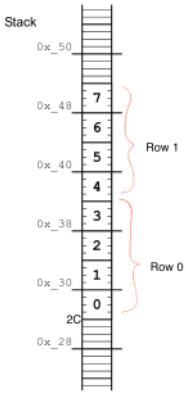
❋ `m` *and* `*m` *are*

❋ `m[i][j]`

2D Arrays 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*
are laid out in a row-major as a single contigious block of memory with one row after another. Unlike the heap one where it can jump around everywhere.

Stack & Heap 2D Array Compatibility

reading from it

→ 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 to the start of row i.

   type? int*
   scale factor? to skip to next element
   SAA is 4 bytes to next column
   HEAP is 4 bytes to next row.

3. `m[0]`? `m[i]`?

4. `m`?

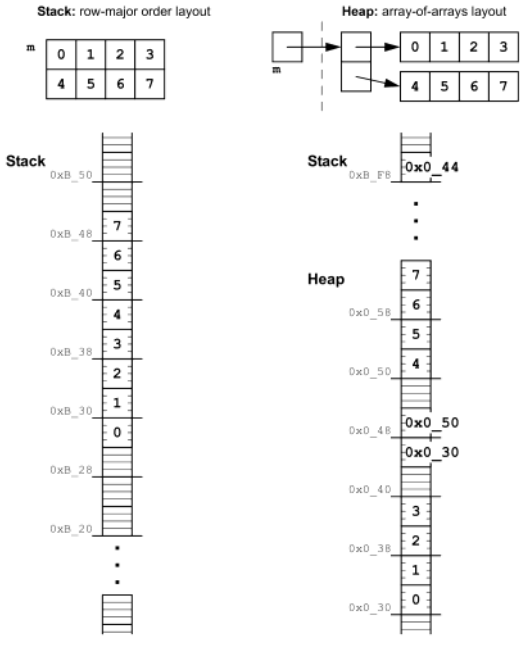
   type?
   scale factor?

For 2D STACK Arrays ONLY

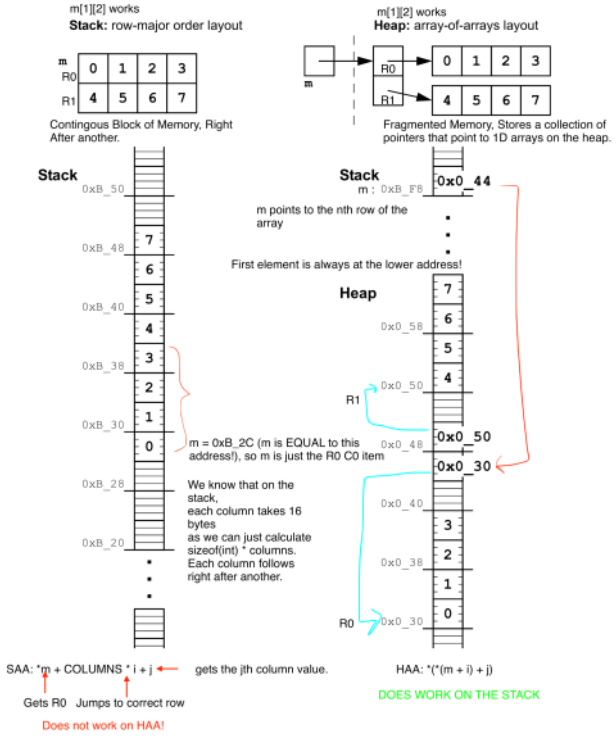
❋ `m` *and* `*m` *are*

❋ `m[i][j]`

2D Arrays: Stack vs. Heap



2D Arrays: Stack vs. Heap



Array Caveats

- *Arrays have no bounds checking!*

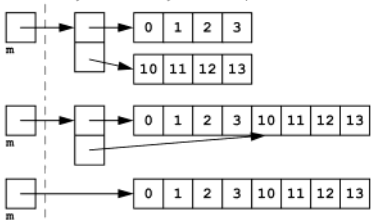
```
int a[5];
for (int i = 0; i < 11; i++)
   a[i] = 0;
```

- *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?

Array Caveats

- *Arrays have no bounds checking!*

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

Everything here compiles fine, since none of this is syntactically wrong. But this is bad, as it will either result in a segfault or interm. error.

- *Arrays cannot be return types!*

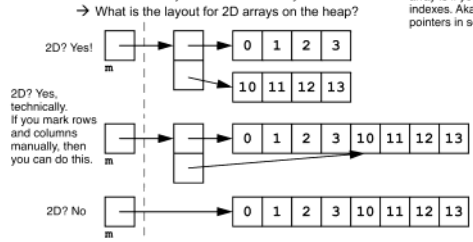
```
int[] makeIntArray(int size) {
   return malloc(sizeof(int) * size);
}
```

You cannot do int[], you must do int* as your return type, and make sure that your returned array is on the heap as a stack one will get cleared!

- *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?

The way something is a 2D array is if you can use 2 indexes. Aka, you need 2 pointers in some way.



All of these are on the heap, as m is pointing to a collection of rows.

- *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)
```

You can also do a[][4] = {{}, {}}; C can figure out the rows.
.length does not exist :(

→ 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)
```

Yes, makes a new a.
Yes, premature function ending though.
Yes, C can infer the rows from the var.
No, type check fails (compiler).
No, we need columns.
Yes, but why.
No!!! A SAA is not an array of pointers!!!

→ Why is all but the first dimension needed?

The compiler only needs the number of columns to compute the start of each row.

Meet Structures

What? A *structure*

-
-
-
-

Why?

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.

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;
```


Meet Structures

What? A *structure*

- Define a new data type.
- Describes a compound unit of storage with data members who can change.
- Accessed using its identifier and data member name.
- When allocated, it is a contiguous fixed size block of memory.

Why?

Enables organizing complex data in a single module.
This is not OOP, since structures cannot contain code (unless you use function pointers but still no polymorphism)

How? Definition

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

Creates a local struct in the local file.

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

More of a global type, that other files can use.

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

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

You can use both, since they are accessed differently.

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

Date d;
```

How? Declaration

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

```
struct Date today;
today.month = 9;
today.day = 22;
today.year = 2022;
```

```
Date today;
today.month = 9;
today.day = 22;
today.year = 2022;
```

dot operator: Does member selection.

- *A structure's data members* Are uninitialized by default, junk data.
- *A structure's identifier used as a source operand* reads the entire structure in.
- *A structure's identifier used as a destination operand* Writes an entire struct.

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

Copies values from today to tomorrow, a true deep copy.
This is slow…so you should probably make tomorrow a struct pointer, and pass today as a reference.

Nesting in Structures and Array of Structures

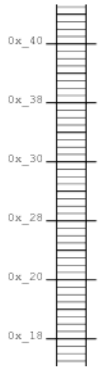
Nesting in Structures

→ 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;


} 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.

Nesting in Structures and Array of Structures

Nesting in Structures

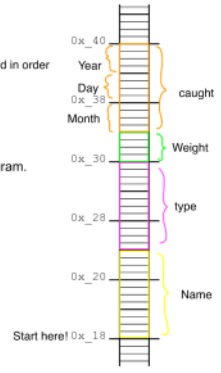
→ Add a `Date` struct, named `caught`, to the structure code below.

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

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

The data members in a struct are stored in order of declaration!

❋ *Structures can contain* other structs and arrays nested as deeply as you want.



→ Identify how a `Pokemon` is laid out in the memory diagram.
It starts at addr. 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, 2022}},
  {"Zayn", "Gay", 153.0, {4, 15, 2003}}
};
```

→ 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) {
```

❋ *Structures are passed-by-value to a function,* COPIES 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 the address (least)

Passing Structures

→ Complete the function below so that it displays a `Date` structure.
typedef Date version from prev page.

```
void printDate (Date date) {
  printf("%i/%i/%i", date.month, date.day, date.year);
```

❋ *Structures are passed-by-value to a function,* which copies entire struct, super 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) {
```

❋ *Recall: Arrays are passed-by-value to a function,*

1

Pointers to Structures

Why? Using pointers to structures

- avoids copying overhead from pass-by-value
- allows called func to change structs whose are passed
- enable heap allocation of structs
- enables efficient linked DS

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 pml = {"Abra","Psychic",30,{1,21,2017}};
  printPm(&pml)
```


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 pml = {"Abra","Psychic",30,{1,21,2017}};
  printPm(  pml)
```