CS 2113
Software Engineering

Lecture 3:
Files and Data Structures in C

Professor Tim Wood - The George Washington University

Previously...

- C Memory
  - Stack and Heap
  - Pointers

FYI -- slides get updated after class

- Memory Management HW
  - Make sense?
  - Better/worse than programming?

- First Lab session
  - File IO and Strings

Write your own code

- You may talk about exercises and projects with other students
- But you must write all of your own code
- If you don't, you will fail the assignment
  - you'll probably fail the exams too!
  - and you might fail at life!

- Remember: talking about how to plan the code is OK. Using identical code, or just changing variable names is not.

This Time

- Memory HW Review
- Dynamic memory allocation from the Heap
- Data struct(ure)s
- Strings and Files (if we have time)

Ex-2 Worksheet

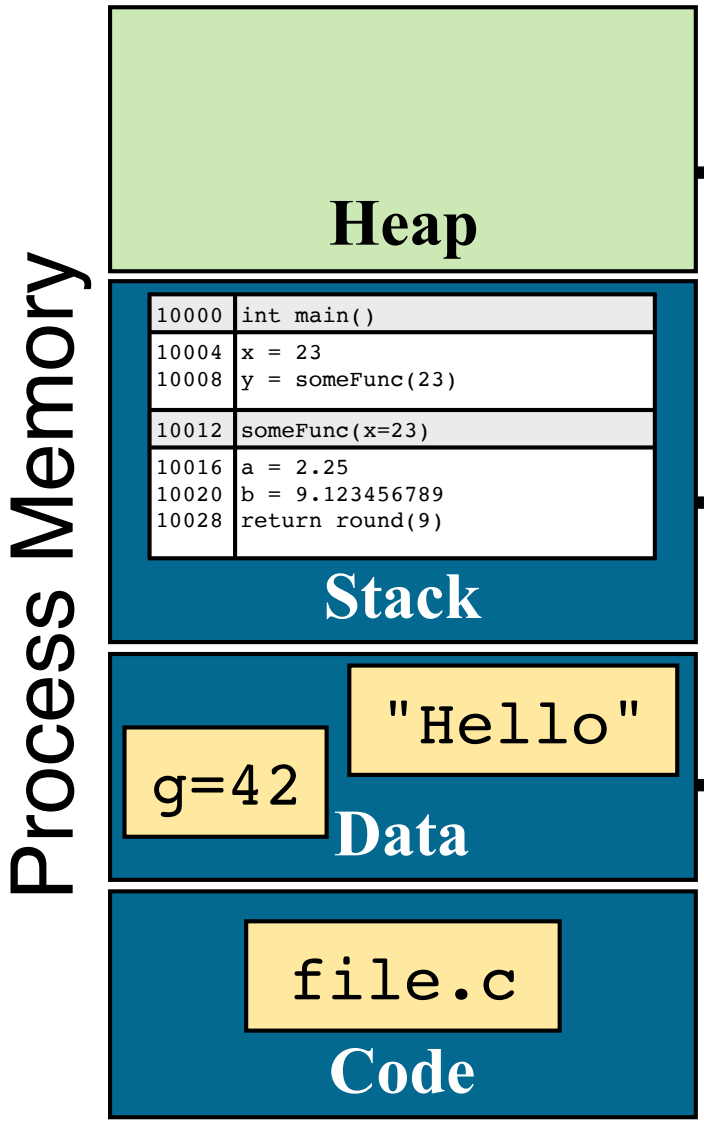
- Question 1

```
i = 42;
c = 't';
pi = &i;
j = *pi;
i = 10;
pc = &c;
*pc = 'X';
```

	Globals	
Address	*Name*	*Contents*
500	i	10
	Stack	
Address	*Name*	*Contents*
10000	j	42
10008	c	X
10009	pi	500
10017	pc	10008
10025	k	???

```
printf("j = %d", j);          _____42_____
printf("pi = %d", pi);        _____500_____
printf("*pi = %d", *pi);      _____10_____
printf("&j = %d", &j);        _____10000_____
printf("&pi = %d", &pi);      _____10009_____
printf("c = %c", c);          _____X_____
```

Heap Segment



Dynamically created memory

Structured area for local variables

Global variables, constants
Easily referenced

The Heap / Memory Pool

- The stack enforces very specific ordering
  - "Local variables" in a function must be on stack!
  - Stack changes with every function call
- Heap is a bit... messier
  - Gives access to variable size chunks of memory
  - Structure is not affected by function calls
  - Required for dynamically sized objects

vs

Aside:
"The heap" and "a heap" (data structure) are not particularly related to one another

images from wikimedia commons

Memory Allocation in C

- Heap acts as a pool of memory for data structures
  - **`ptr=malloc(size);`**
    - Ask the heap for **`size`** bytes of memory
    - Returns a memory address (so assign it to a pointer!) or 0 if the allocation failed (so check it!)
  - **`free(ptr);`**
    - Release the memory when you are done so it can be reused

```
#include <stdlib.h> // include to use malloc without warnings
#include <stdio.h>
int main()
{
  int *myInt;
  myInt = (int*) malloc(4); // request 4 bytes for an integer
  if(myInt == NULL) { return -1;}
  *myInt = 1234567;
  printf("Value of myInt = %d\n", *myInt);
  free(myInt); // release memory when done
  return 0;
}
```

Stack + Heap

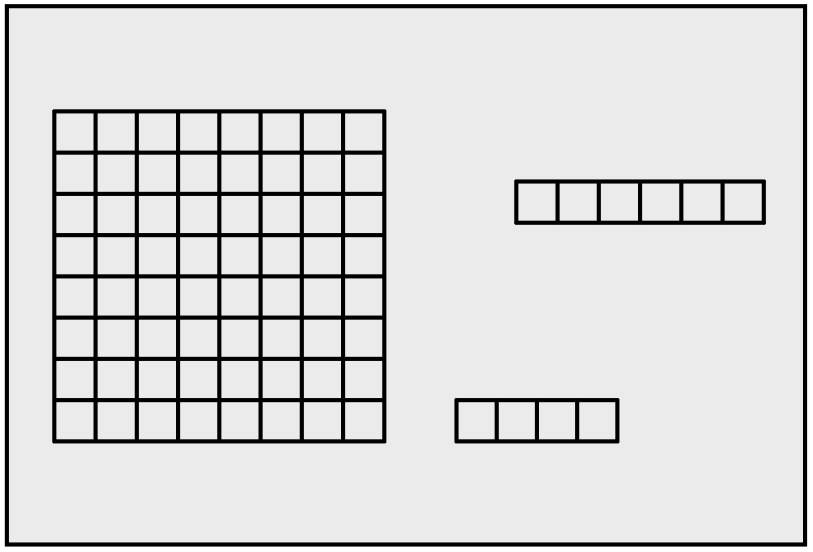
- Need a pointer to reach the heap

```
int main()
{
  int *myInt;
  double *myDouble;
  char *myString;
  myInt = (int*) malloc(4);
  myDouble = (double*)malloc(8);
  myString = (char*)malloc(64)
  ...
}
```

Stack

10000	int main()
10004	*myInt --> address 99048
10008	*myDouble --> address 99743
10012	*myString --> address 98321

Heap



- Pointer addresses stored in the stack
  - "Value" of pointer is kept on the heap

How much space?

- When you use malloc, you need to specify the exact size you need
- Usually based on type of data you are storing

```
int *myInt;
int *myInt2;
myInt = (int*) malloc(4); // request 4 bytes
myInt2 = (int*) malloc(sizeof(int)); // request enough bytes for an integer
```

- Which is better code?

(Simplified) Memory Layout

```
void firstFunc() {
    int a = 20;
    int b = 30;
    int *p;  p = (int*) malloc(sizeof(int));
    *p = 5;
    b = *p;
    free(p);
    secondFunc();
    p = malloc(sizeof(int));
    *p = 50;
    /* DRAW MEMORY @ THIS LINE */
}
void secondFunc() {
    int *q = (int*) malloc(sizeof(int));
    int *r = (int*) malloc(sizeof(int));

    *q = 45;
    *r = 100;
    free(q);
    return;
}
int main() {
    int x = 10;
    int *y = &x;
    firstFunc();
    return 0;
}
```

Assume:
- int and int* use 4 bytes
- No extra space needed for function headers

Stack

Address	*Name*	*Contents*
10000		
10004		
10008		
10012		
10016		
10020		
10024		
10028		
10032		

(addresses grow up)
start: 10,000
Stack

Heap

Address	*Name*	*Contents*
50000		
49996		
49992		
49988		
49984		
49980		
49976		
49972		
49968		

start: 50,000
(addresses grow down)
Heap

(Simplified) Memory Layout

```
void firstFunc() {
    int a = 20;
    int b = 30;
    int *p = malloc(sizeof(int));
    *p = 5;
    b = *p;
    free(p);
    secondFunc();
    p = malloc(sizeof(int));
    *p = 50;
    /* DRAW MEMORY @ THIS LINE */
}
void secondFunc() {
    int *q = (int*) malloc(sizeof(int));
    int *r = (int*) malloc(sizeof(int));

    *q = 45;
    *r = 100;
    free(q);
    return;
}
int main() {
    int x = 10;
    int *y = &x;
    firstFunc();
    return 0;
}
```

Stack

Address	*Name*	*Contents*
10000	x	10
10004	*y	10000
10008	a	20
10012	b	5
10016	*p	50000
10020	*q	50000
10024	*r	49996
10028		
10032		

Heap

Address	*Alloc?*	*Contents*
50000	YES	50
49996	YES	100
49992	NO	
49988	NO	
49984	NO	
49980	NO	
49976	NO	
49972	NO	
49968	NO	

The Heap Sticks Around

- If you call **`malloc`**, you must also call **`free`** at some point
  - Need to clean up data structures you are no longer using
- **Memory Leak**: wasting resources by not freeing data after the program is done with it

```
int* myInt;
for(i=0; i < 1000000; i++)
{
  myInt = (int*) malloc(4*1000);
  // do something with array
}
free(myInt);
```

- Be sure to balance **`malloc`** / **`free`** calls!

Dynamic Arrays

- Pointers can also be used like arrays
  - Remember: allocate space in bytes
    - Use **`sizeof(TYPE)`** to find size of a var type in bytes
  - Can index into pointer using array[] syntax
    - Do not need to use dereference operand!

```
int earthDays = 365;
int jupiterDays = 10563;

int* days;
days = (int*) malloc(sizeof(int)*earthDays);
days[32] = 1; // set my birthday
// ...
free(days);

days = (int*) malloc(sizeof(int)*jupiterDays);
days[4632] = 1;
```

Pointers and Pointers

- What does this do?

```
int a = 1;
int b = 5;
int *x;
int *y;

x = &a;
*x = 100;

y = x; //////
*y = 1000;
x = &b;
*x = 2000;

printf("a=%d  b=%d \n", a, b); /// 1000  2000
printf("*x=%d  *y=%d \n", *x, *y);  //  2000 1000
```

a	1000
b	2000
*x	&b
*y	&a

Pointers and Pointers

- What does this do?

```
int a = 1;
int b = 5;
int *x;
int *y;

x = &a;
*x = 100;

y = x;          // y points to the same address as x
*y = 1000;
x = &b;
*x = 2000;      // this has no impact on y

printf("a=%d  b=%d \n", a, b);          //  a=1000  b=2000
printf("*x=%d  *y=%d \n", *x, *y);      // *x=2000 *y=1000
```

Pointer Math

- Can treat arrays as pointers and vice versa

```
int nums[] = {1, 3, 5, 7, 9};
printf("%d ", nums[2]);
printf("%d ", *(nums+2);
```

- The address in a pointer is just an int
  - You can use math with ints! and with pointers!
  - If you add X to a pointer, the address is changed by:

X*sizeof(ptr data type)

```
int* dayArray = (int*) malloc(sizeof(int)*365);
*dayArray = 1; // set first entry in array
dayArray += 1;  // adjust address by 1 entry
*dayArray = 2; // set second entry in array
dayArray += 500; // beyond the end of the array!
```

Pointer Math Fun

- Run this code. How will the array end up?

0	1	2	3	4	5	6	7	8	9

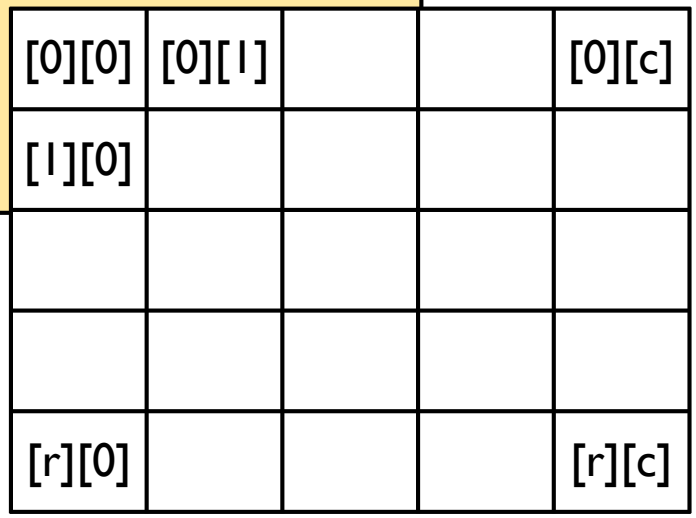
```
int* array = (int*) malloc(sizeof(int)*10);
// fill array with the values 0...9

array[3] = 20;
*array = 100;
array += 1;
*array = 2;
array[3] = 60;
array = array + 2;
array[0] = array[0] + 2;
*(array + 3) = 90;
```

2D arrays

- We can have arrays of strings (char[])

```
char text[500][100]
// 500 rows, each with an array of 100 chars
text[10][0] = 'a';
strcpy(text[14], someString);
```



- Or any other data type

```
int array2d[10][20];
for(r=0; r < 10; r++)
 for(c=0; c < 20; c++)
   printf("array2d[%d][%d]=%d\n",r,c,alottaInts[r][c]);
```

Or an array of pointers

- We can have arrays of strings (char*)

```
char* text[500]
// 500 rows, each with variable size
text[1] = malloc(20);
text[2] = malloc(100);
text[3] = malloc(2);
```

- What does the memory layout look like?

- Why would you do this?

Structures

Professor Tim Wood - The George Washington University

Advanced Data Types

- In Java you could easily define objects:
  - Multiple pieces of data
  - Private / Public functions
  - Subclasses and inheritance

```
public class Employee {
  private String name;
  private double salary;
  ...
  public double getTaxRate(){
  ...
  }
```

Java

- C has minimal support for objects
  - Can combine multiple pieces of data
  - Cannot have functions inside an "object"

```
struct employee {
  char* name;
  double salary;
};

double getTaxRate(struct employee e) {
...
}
```

C

Structs

```
struct employee{
  char *name;
  double salary;
};

int main() {
  struct employee boss;
  boss.name = "B. Gates";
  boss.salary = 1000000000;
...
}
```

- **`struct`** = composite data structure
- **`employee`** = name of our new data type
- Must use full "**`struct employee`**" when declaring

Why use structs?

- Data **encapsulation**
  - Much easier than having lots of variables for each component
  - Which is cleaner?

```
void eat(char *foodName, int cal, int protein, int fat, int vitC)

void eat(struct nutritionInfo *food);
```

Usually you will want to use pointers to structs--otherwise they will be copied by value to the function!

Structs and Pointers

- More annoying syntax to learn!
- Access a struct variable's internals with <dot>
- Access pointer to a struct with ->

```
struct employee boss;
struct employee *coderPtr = malloc(sizeof(struct employee));

boss.name = "B. Gates";
boss.salary = 1000000000;

coderPtr->name = "Code Monkey";
coderPtr->salary = 50000;
```

The arrow is a shortcut for `(*coderPtr).XXX`

- If you do it backwards you get a compiler error:

```
error: invalid type argument of ‘->
error: request for member ‘name’ in something not a
structure or union
```

Structs

- Define a struct to store information about a place:
  - its name (as a char*)
  - its latitude (as a float)
  - its longitude (as a float)
- Instantiate two of your structs:
  - one as a pointer, one as a regular variable
- Fill in the 3 members of each struct
  - for example: "DC", 38.889404, -77.035194
- Compile and get it to work on CodeAnywhere
- **Put your answers into the quiz on blackboard**

```c
#include <stdlib.h>
#include <stdio.h>

struct location {
  char *name;
  float lat;
  float lon;
};

int main() {
  struct location myHouse;
  struct place *DC;
  DC = (struct location*)malloc(sizeof(struct location));
  DC->name = "Washington DC";
  DC->lat = 38.889404;
  DC->lon = -77.035194;

  myHouse.name = "Prof. Wood's Igloo";
  myHouse.lat = 77.828029;
  myHouse.lon = -88.057823;

  printf("I live at %f, %f\n", myHouse.lat, myHouse.lon);
  printf("I commute to %f, %f\n", DC->lat, DC->lon);

  return 0;
}
```