C Pointers Workshop

Max Petrushin

Objectives

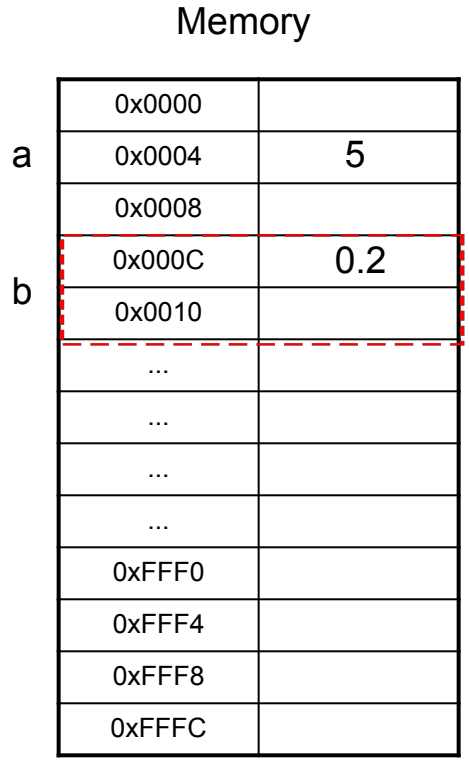
- Learn C pointer rules & syntax
- Understand advanced pointer concepts (array, structure, pointers to pointers)
- Understand why C pointers are important (use cases for pointers)
- Go over example problems

C Pointer Rules

1) Getting address of a variable

- &(variable_name) // notation to get address of variable
- Ex:
  - int a = 5; &a // address of integer a
  - double b = 0.2; &(b) // address of double b

- print:
  - printf("%p", &a); // %p short for pointer (hex format)
  - printf("%d", &(a)); // %d prints address in decimal format



C Pointer Rules

2) Dereference - going from address to variable (content of that address)

- *(&(variable_name)) // notation to dereference the address of variable
- Ex:
  - *(&a) // Dereferencing the address of a
  - *(&b) // Dereferencing the address of b

- print:
  - printf("%d", *(&a)); // prints value of a (5)
  - printf("%f", *(&b)); // prints value of b (0.2)

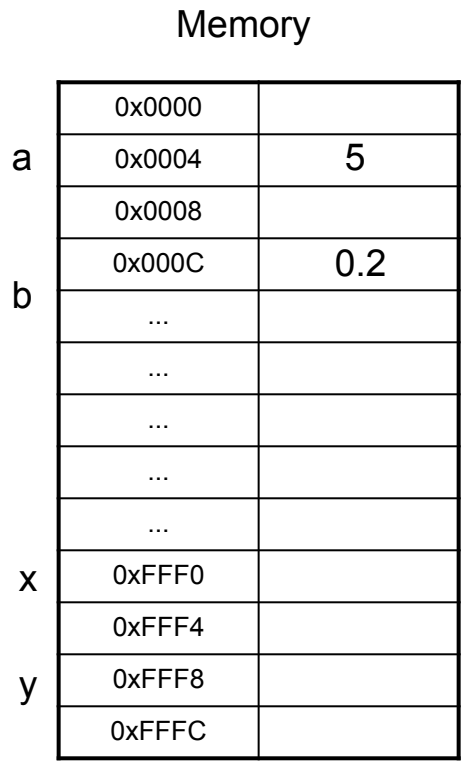
Memory

	0x0000	
a	0x0004	5
	0x0008	
	0x000C	0.2
b	...	
	...	
	...	
	...	
	...	
	0xFFF0	
	0xFFF4	
	0xFFF8	
	0xFFFC	

C Pointer Rules

3) Pointer type (another C data type, just like int, double, char, etc…)

- data_type* variable_name; // notation to define a pointer variable
- Ex:
  - int* x; // Integer pointer x
  - double* y; // Double pointer y



C Pointer Rules

4) Pointer type == Address of a variable

- Pointer type exists to store the address of the variable (just like integer type exists to store integers)
- Ex:
  - int a = 5; // integer a
  - int* x; // integer pointer x
  - x = &a; // integer pointer x stores address of integer a\
  - print(“%d”, *x); // prints “5”

  - double b = 0.2; // double b
  - double* y; // double pointer y
  - y = &b; // double pointer y stores address of double b
  - print(“%f”, *y); // prints “0.2”

Memory

	0x0000	
a	0x0004	5
	0x0008	
	0x000C	0.2
b	...	
	...	
	...	
	...	
	...	
x	0xFFF0	0x0004
	0xFFF4	
y	0xFFF8	0x000C
	0xFFFC	

C Pointer Rules

Many meanings of * (star)

- Multiplication
  - int x = 2 * 3; // x is assigned to 6 (2*3)

- Pointer data type indication
  - int* y; // we use * to replace word “pointer” in “integer pointer”
  - float* z; // we use * to replace word “pointer” in “float pointer”

- Dereferencing
  - y = &x; // pointer y is assigned to address of x
  - *(y); // dereferencing pointer y, gives you “6”
  - *(&x); // dereferencing the address of x, gives you “6”

C Pointer Rules

Declaration vs. Initialization of Pointers

- int x = 5;
- int* a; // Declaration of pointer a
- a = &x; // Assigning …

- int x = 5;
- int* a = &x; // Initialization of pointer a

They both do the same thing. Both are valid.

Advanced Pointer Concepts

1) Pointer to Pointers

- Before, we looked at Pointers to Variables!
- Now, we're looking at Pointers to Pointers … (Double-Pointer)
  - int x; // declaring an integer x
  - x = 5;

  - int* y; // declaring a pointer to integer
  - y = &x; // assigning it to address of integer

  - int** z; // declaring a pointer to a pointer to integer
  - z = &y; // assigning it to address of …

Memory

	0x0000	
x	0x0004	5
	0x0008	
y	0x000C	0x0004
	...	
	...	
	...	
	...	
	...	
z	0xFFF0	0x000C
	0xFFF4	
	0xFFF8	
	0xFFFC	

Advanced Pointer Concepts

1) Pointer to Pointers

- Dereferencing double-pointers
  - printf(“%d”, x); // prints “5”
  - printf(“%d”, *y); // prints “5”
  - printf(“%d”, **z); // prints “5”

  - printf(“%p”, z); // prints “0x000C”
  - printf(“%p”, &y); // prints “0x000C”
  -
  - printf(“%p”, y); // prints “0x0004”
  - printf(“%p”, &x); // prints “0x0004”

Memory

	0x0000	
x	0x0004	5
	0x0008	
y	0x000C	0x0004
	...	
	...	
	...	
	...	
	...	
z	0xFFF0	0x000C
	0xFFF4	
	0xFFF8	
	0xFFFC	

Advanced Pointer Concepts

2) Pointer Arithmetic

- x = x + 2; // Regular int arithmetic, x is now “7”
- y = y + 1; // Pointer arithmetic, y is now “0x0008”

How big is increment (or decrement)? Depends on what it’s pointing to!

- 4 bytes - int // y++; makes y equal to “0x0008”
- 8 bytes - double // y++; makes y equal to “0x000C”
- 1 bytes - char // y++; makes y equal to “0x0009”
- Depends on a machine too (32bit vs 64bit etc…)

Memory

	0x0000	
x	0x0004	5
	0x0008	
y	0x000C	0x0004
	0x0010	
	...	
	...	
	...	
	...	
	0xFFF0	
	0xFFF4	
	0xFFF8	
	0xFFFC	

Advanced Pointer Concepts

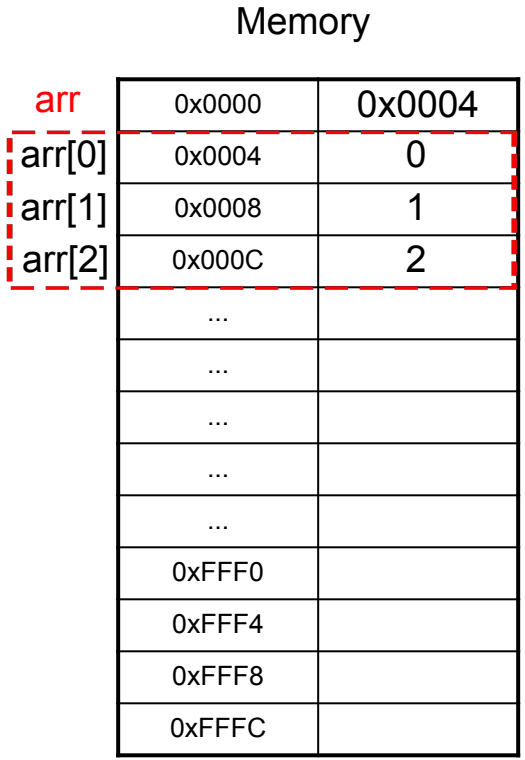
3) Arrays and Pointers

Array - variables of same type continuously stored in a single block

- int arr[3] = {0, 1, 2}; // Initializing array of size 3

Array variables name is actually a pointer to it's first element! Just remember this…

- arr == &arr[0] // Just remember this…
- arr // This is a pointer to arr[0]
- arr+1 // This is a pointer to arr[1]
- arr+2 // This is a pointer to arr[2]



Advanced Pointer Concepts

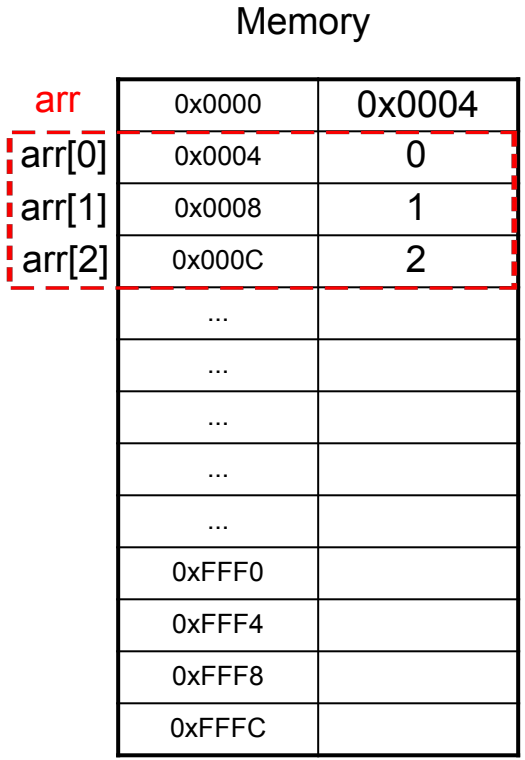
3) Arrays and Pointers

Accessing array elements through pointer arithmetic

- *(arr) // “0”
- *(arr+1) // “1”
- *(arr+2) // “2”

Is equivalent to:

- arr[0] // “0”
- arr[1] // “1”
- arr[2] // “2”



Why C pointers are Important?

1) Pointers allow us to manipulate (change) variable values from any scope (function) of the program

Pass By Value

```c
#include <stdio.h>
void doubleValue(int val);

int main(){
  int a = 5;
  doubleValue(a);
  printf("%d\n", a); // Prints "5"
  return 0;
}

void doubleValue(int val){
  val = val * 2;
}
```

Pass By Reference

```c
#include <stdio.h>
void doubleValue(int* val);

int main(){
  int a = 5;
  int* b;
  b = &a;
  doubleValue(b);
  printf("%d\n", a); // Prints "10" !!!
  return 0;
}

void doubleValue(int* val){
  *val = *val * 2;
}
```

Why C pointers are Important?

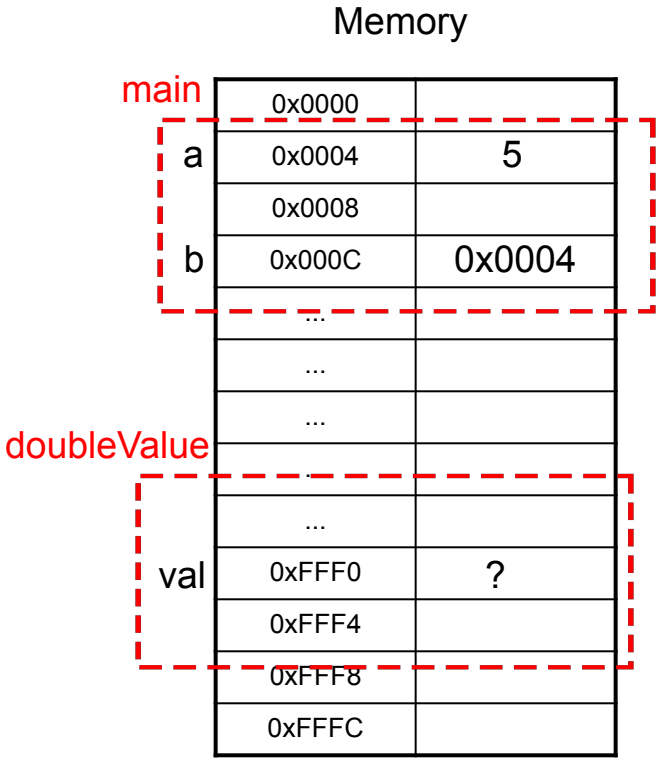
1) What are we passing in both scenarios?

Pass By Value

```
11    void doubleValue(int val){
12      val = val * 2;
13    }
```

Pass By Reference

```
13    void doubleValue(int* val){
14      *val = *val * 2;
15    }
```



Why C pointers are Important?

2) Memory allocation/reservation! Computer doesn't know how much memory your arrays and structures need. Sometimes even you don't know it!

- We need a way to request X amount of memory from computer, at the time we learn what our X is (at a dynamic time).
- There is a tool (function) that helps with it. It is called malloc (memory allocate)
  - Malloc makes a call to Operating System (OS)
  - OS decides which "chunk" of memory to "reserve" for our program
  - OS tells us which exact "chunk" it reserved for us by returning us a **pointer** to that "chunk"
  - When we are done using that memory "chunk", we must **free** that "chunk" otherwise we will eventually run out of the usable memory (memory leak)

Why C pointers are Important?

2) Memory allocation/reservation! (and then releasing/free-ing)

- int* arr = (int *) malloc(23 * sizeof(int)); // allocates enough memory for an integer array of size 23 elements
- free(arr); // frees the memory

- Match every malloc with a free
- Always!
- Don't forget #include <stdlib.h>

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

int main(){
  int X;
  int* array;
  printf("Enter X (size of array): ");
  scanf("%d", &X);

  // Requesting to allocate memory for X sized array
  array = (int *) malloc(X * sizeof(int));

  for (int i=0; i<X; i++){
    array[i] = i+1;
    printf("%d, ", array[i]); // Prints 1, 2, ... , X
  }
  printf("\n");
  free(array);  // Requesting to free/release the allocated memory
  return 0;
}
```

Why C pointers are Important?

3) Working with Structures

- Elements of the structure can be pointers (can be allocated dynamically)

```c
typedef struct User
{
    int id;
    int *friend_id_list;
} User;
```

```c
User user;
user.friend_id_list = malloc(sizeof(int) * 2);
user.id = 5;
user.friend_id_list[0] = 0;
user.friend_id_list[1] = 1;
free(user.friend_id_list);
```

- Structures as a whole can be pointers and can be dynamically allocated with malloc!

```c
User *users;
users = malloc(sizeof(User) * SIZE);
for (int i=0; i < SIZE; i++){
  users[i].id = i;
  users[i].friend_id_list = malloc(sizeof(int) * 2);
}
```

```c
for (int i=0; i < SIZE; i++){
  free(users[i].friend_id_list);
}
free(users);
```

Why C pointers are Important?

3) Working with Structures

- Accessing structure array elements:
  - users[1].id // result is “1”
  - (*(users+1)).id // result is “1”

- -> (arrow) notation
  - (users+1)->id // result is “1”

-> notation does 2 things in 1:

- dereference the pointer
- . (dot) to access the field of the structure (id in our case)

Example Problems

Board work (in IDE) solving pointer problems