week 4_3:

pointers

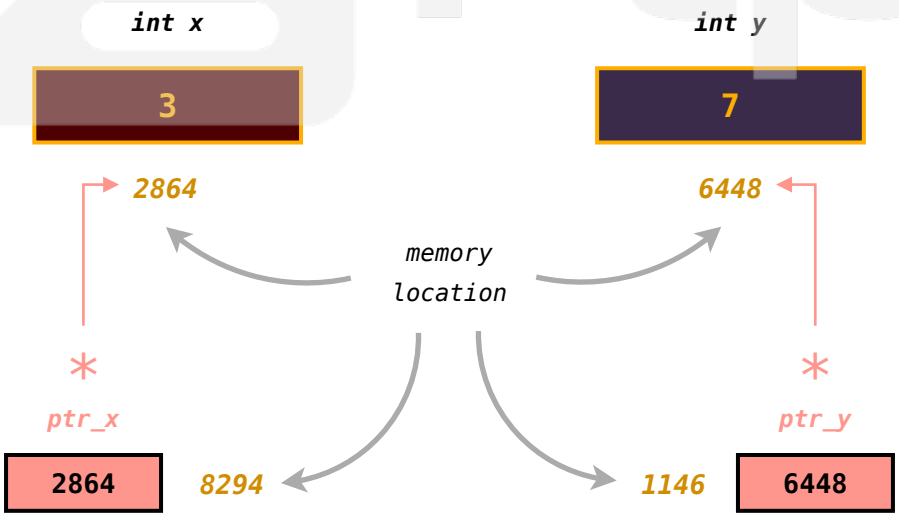
rajin teaches programming

INTRODUCTION TO POINTERS:

Pointers are one of the **most powerful** features of the C programming language**.** They provide a means to directly **access** and **manipulate memory,** which can lead to more efficient and flexible code**.** Understanding pointers is crucial for effective C programming, as they are used in a variety of contexts including **arrays, functions,** and **dynamic memory allocation.**

– WHAT ARE POINTERS?

A **pointer** is a **variable** that stores the **memory address** of **another variable.** Instead of holding a **data value** directly, a pointer holds the **address where** the data is **stored.** This allows for efficient array handling, dynamic memory allocation, and complex **data structures** like **linked lists** and **trees.**



In the visualization above, we have two **integer variables, x** and **y,** and the **pointer variables** store the **memory location/address** of the integer variables, **not** the **value** of the variable itself**.** In turn, each **pointer** variable **has its own memory space/location.**

– HOW DO POINTERS WORK IN MEMORY?

In **memory,** each **variable** is **stored** at a **specific address.** A **pointer variable** holds the **address** of **another variable,** enabling **direct access** to that **memory location.** When you work with pointers, you **manipulate** the **memory address** rather than the **value stored** at that address.

DECLARING AND USING POINTERS:

POINTERS

```
type *pointerName; //Declaration
pointerName = &variable; //Initialization with an address
```

To **declare** a pointer, you use the `*` **operator.** `type` is the **data type** of the variable that the pointer **points** to. `pointerName` is the **name** of the **pointer.** `&` is the **address-of operator,** which gives the **address** of `variable`.

– POINTER DEREFERENCING:

POINTERS

```
// after initialization
printf("value: %d", *pointerName);
```

Only after declaration and initialization, `*` is used to **access** the **value** at the **address stored** in `pointerName`. We say that `*` now acts as a **dereferencing pointer.**

Developer - pointer.c

```c
#include<stdio.h>

int main() {

    int var = 10;
    int *ptr;

    ptr = &var;

    printf("\nValue of var: %d\n", var);
    printf("Address of var: %p\n", &var);

    printf("\nValue of ptr: %p\n", ptr);
    printf("Value pointed to by ptr: %d\n", *ptr);

    return 0;
}
```

variable declared & initialized

pointer declared

pointer initialized

pointer dereferenced

an example of pointer usage.

So, say you want to find out the value stored in the variable **var**. You could either refer to the variable itself, or dereference the pointer that is pointing to it, i.e, ***ptr**.

Developer - pointerInLoop.c

```c
#include <stdio.h>

int main() {

    int arr[3] = {1, 2, 3};

    int *p = arr;

    for (int i = 0; i < 3; i++) {

        printf("Value at arr[%d]: %d\n", i, *(p + i));
    }

    return 0;
}
```

another example of pointers being used to dereference in loops.

COMMON USE CASES:

- DYNAMIC MEMORY ALLOCATION:

Pointers are **essential** for **dynamic memory allocation** using `malloc`, `calloc`, and `free` functions.

- FUNCTION PARAMETERS:

Pointers allow **functions** to **modify** variables **outside** their **scope**, enabling **pass-by-reference** semantics.

- BEST PRACTICES:

Always **initialize pointers** to **NULL** if they are **not** assigned any address. Avoid **dereferencing NULL** or **uninitialized** pointers to prevent **runtime errors.** Use pointers **judiciously** to avoid **complexity** and potential memory leaks.

SUMMARY:

Pointers store **memory addresses** of **variables.** They enable **direct memory access** and **manipulation.** Essential for **dynamic** memory allocation, function **parameter passing**, and efficient **array handling.** Proper use of pointers can lead to **efficient** and **flexible** code.

SOME FAQs:

What is a NULL pointer?

A NULL pointer is a pointer that does not point to any memory location. It is often used for pointer initialization.

How do you avoid pointer-related errors?

- Initialize pointers to NULL.
- Avoid dereferencing uninitialized or NULL pointers.
- Use pointers judiciously and understand their scope and lifetime.

Developer - array.c

```c
#include <stdio.h>

int main() {

    int arr[5] = {10, 20, 30, 40, 50};
    int *p = arr;  // Pointer to the first element of the array

    for (int i = 0; i < 5; i++) {

        printf("arr[%d] = %d\n", i, *(p + i));  // Accessing array elements using pointer
    }

    for (int i = 0; i < 5; i++) {

        *(p + i) += 10;  // Modifying array elements using pointer
    }

    for (int i = 0; i < 5; i++) {

        printf("Modified arr[%d] = %d\n", i, arr[i]);
    }

    return 0;
}
```

a program using pointers to access, modify, and print from arrays

a program making use of pointers
for passing by reference

CONCLUSION:

By following these guidelines and understanding the examples provided, you should have a comprehensive understanding of pointers in C programming. Pointers are a powerful tool that, when used correctly, can greatly enhance the efficiency and flexibility of your code.

next class 4_3_1:

pointer problems

rajin teaches programming

reach out for classes at: rajin.khan2001@gmail.com