

CIT 5950 Recitation 1 - C, Pointers, and Codio

Welcome to recitation!!! 😊

Pointers

Pointers are a data type that store a memory address. We use them for a number of things in C, such as:

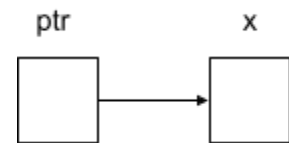
- Simulating “pass-by-reference”
- Using function arguments as return values (also known as “**output parameters**”)
- Avoiding copying whole data structures when passing arguments into functions

If we have a variable x , then $\&x$ will give us the address of x . If we have a pointer p , $*p$ will give us the value stored at the address p is holding, or “the value p points to.”

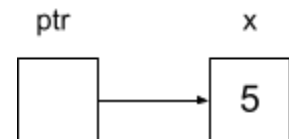
Let’s look at an example!

```
int32_t x;  
int32_t *ptr;  
ptr = &x;  
x = 5;  
*ptr = 10;
```

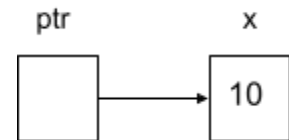
1) We can represent the result of the above three lines of code graphically. ptr stores the address of x . It “points to x .” x currently doesn’t have a value because we did not assign it one!



2) After executing $x = 5$, our diagram changes.



3) After executing, $*ptr = 10$, our diagram changes again. Notice that x has been modified by dereferencing ptr .



Exercise 1:

Draw a memory diagram like the one above for the following code and determine what the output will be.

```
void foo(int32_t *x, int32_t *y, int32_t *z) {  
    x = y;  
    *x = *z;  
    *z = 37;  
}  
  
int main(int argc, char *argv[]) {  
    int32_t x = 5, y = 22, z = 42;  
    foo(&x, &y, &z);  
    printf("%d, %d, %d\n", x, y, z);  
    return EXIT_SUCCESS;  
}
```

The following code has a bug. What's the problem, and how would you fix it?

```
void bar(char *str) {
    str = "ok bye!";
}

int main(int argc, char *argv[]) {
    char *str = "hello world!";
    bar(str);
    printf("%s\n", str); // should print "ok bye!"
    return EXIT_SUCCESS;
}
```

Output Parameters

As you can see in the above examples, pointers let us modify the parameters we pass in (more precisely, we can modify the data our argument points to). This leads us to a special kind of parameter known as an **output parameter**. As the name suggests, this refers to a parameter that we use to store an output of a function. These are very common in C and you will see a lot of library functions that use these.

Exercise 2:

`strcpy` is a function from the standard library that copies a string `src` into an output parameter called `dest` and returns a pointer to the beginning of the destination string. Write the function below. You may assume that `dest` has sufficient space to store `src`.

```
char *strcpy(char *dest, char *src) {

}

}
```

How is the caller able to see the changes in `dest` if C is pass-by-value?

Why do we need an output parameter? Why can't we just return an array we create in `strcpy`?

Pointer Arithmetic and Arrays

We can do addition and subtraction to pointers, with a catch. Arithmetic on pointers is scaled to the size of the type being pointed to (in bytes). So, in the above example, `ptr + 1` would actually increase the value of `ptr` by 4 since it points to a 32-bit integer.

Arrays and pointers are very closely related. Array subscript notation is just special syntax for pointer arithmetic with `arr[i]` being equivalent to `*(arr + i)`. Using an array name in an expression returns the address of the first element in the array.

Exercise 4 (Bonus):

Given the following command: `mkdir -v cats dogs` and `argv = 0x1000`, draw a box-and-arrow memory diagram of `argv` and its contents for when `mkdir` executes.



Using the same information from above, what can you say about the values returned by the following expressions? You may not be able to tell the exact value returned, but you should be able to describe what that value is/represents.

- 1) `argv[0]`
- 2) `argv + 1`
- 3) `*(argv[1] + 1)`
- 4) `argv[0] + 1`
- 5) `argv[0][3]`

Exercise 5 (bonus):

A prefix sum over an array is the running total of all numbers in the array up to and including the current number. For example, given the array {1, 2, 3, 4}, the prefix sum would be {1, 3, 6, 10}.

Write a function to compute the prefix sum of an array given a pointer to its first element, the pointer to the first element of the output array, and the length of both arrays (assumed to be the same).

```
void prefix_sum(int *input, int *output, int length) {
```

```
}
```