CIT 5950 Recitation 0 - C, Pointers, and Docker

Welcome to recitation!!! 😃

Pointers

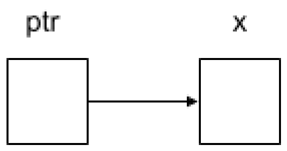
Pointers are a data type that store a memory address. We use them for a number of things in C, such as:

- 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

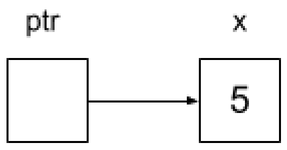
If we have a variable `x`, then `&x` will give us the address of `x`. If we have a pointer `p`, `*p` will give us the value stored at the address `p` is holding, or "the value `p` points to."

Let's look at an example!

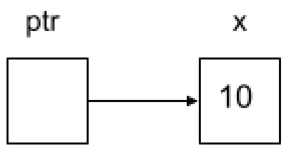
```
  int32_t x;
  int32_t *ptr;
  ptr = &x;
  x = 5;
  *ptr = 10;
```



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`.

<u>Exercise 1a</u>:

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;
}
```

Exercise 1b:
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?

<u>Exercise 3</u>:
More practice with output parameters and arrays.

Write a function to compute the sum of values and product of all values in an array. The function is given a pointer to the first element in an array, the length of the array, and two output parameters to return the product and sum.

```
void product_and_sum(int *input, int length, int *product,
                                                     int *sum) {

























}
```

<u>Exercise 4</u>:
More practice with output parameters, arrays, and the heap.

Write a function to filter values out of a given input array.
This function takes in an array of integers and the array length. The function creates a new array allocated on the heap that is the same as the input array, except that all values that are equal to the input parameter “filter” are not in the array. For example:

```
int arr[] = {5, 9, 5, 0};

int* filtered;

size_t filtered_len = filter(arr, 4, 5, &filtered);

// filtered_len should be 2
// filtered should be {9, 0}
```

Note how the output array is passed via output parameter and how the function just returns the length of the new array.

```
----------------------------------------------------------------

size_t filter(int *input, size_t length, int filter, int** out){









}
```