Tutorial 6

- Memory Addresses and Pointers
- Pointers to Structures
- Function Pointers and Testing
Memory Addresses and Pointers

This is a pointer:

```c
int i = 42;
int *p = NULL;
p = &i;
```

**Address operator** (&) gets the address of a variable in memory.

**Indirection operator** (*) gets the value of what a pointer “points at”.

```c
// To print the value p is pointing at
printf("The value of *p is: %d", *p);

// To print the value of p
printf("The value of p is: %p", p);

// To print the address of p
printf("The address of p is: %p", &p);
```
Pointers as Function Parameters

- Allows functions to mutate variables that live outside the function. (A new side effect!)
- Allows functions to avoid copying large structures.
- Allows functions to "return" multiple values.
Example: Mutating Values in Functions

Compare these two segments of code

```c
void inc(int p) {
    p += 1;
}

int main(void) {
    int x = 42;
    inc(x);
    printf("%d\n", x);
}
```

Output:
42

```c
void inc(int *p) {
    *p += 1;
}

int main(void) {
    int x = 42;
    inc(&x);
    printf("%d\n", x);
}
```

Output:
43
Exercise: q1-func.ptr

You are given the following structure apply_with.

```c
struct apply_with {
    int (*fp)(int, int);
    int x;
    int y;
};
```

Define the following C function:

```c
// eval(s) returns the value of evaluating function s->fp with parameters s->x and s->y.
// requires: s and s->fp are valid pointers.
```
Exercise: q2-testing

Define the following C function:

```c
// test_div(fp) tests a divide function fp by running multiple tests, returns true if fp passes all tests, and false otherwise. // requires: fp is a valid pointer.
```

Define also the following divide functions:

```c
// my_div(i, j) returns the result of i / j. // requires: j != 0.
// my_div_broken(i, j) should return the result of i / j (but fails to do so). // requires: j != 0.
```