Pointers in C

pointy pointy pointy pointy pointy pointy pointy

(These slides use the word ‘ptr’ when describing the object, because saying the “pointer is pointing to a pointer pointing at an int” is too much)

Pointer Declaration in C:

```
type * name;
```

The data type is used to determine the way that ptrs count when doing <u>pointer math</u>, as well as describing the object that ptr is pointing to.

This is the <u>indirection operator</u>. In the declaration of a pointer, it simply demonstrates that you are creating a pointer to a type instead of a variable of that type.

When talking about pointers, the name has no special meaning like with arrays.

Background Understanding

A variable in C can be imagined like a box somewhere in memory that you have given a name. It has three pieces: a **name**, a **value**, and a **memory address**.

So what’s the difference between a normal variable and a pointer? The difference is that a variable holds a value that could be anything at all (including memory addresses). A pointer on the other hand, always thinks that it is holding a memory address.

There are two operators that are especially associated to pointers:

- the reference operator : &
- the dereference operator : * (this is not the same as when used in declaration, hence the blue)

We will cover those soon. For now, just know the names.

Double, Triple, etc. Pointers

The declaration of a pointer does not use * as the dereference operator (which is why I didn't highlight it blue). Instead, this asterisk simply tells you that the data-type in question is a pointer type:

```
        int * a;
        float * b;
```

Since int* is in fact a type of its own, we can use it as the "type" from slide 2; this creates a double pointer:

```
        int* * c;
```

We can, in fact, stack it as many times as we like:

```
        int*** d; // you could put as many * as you'd like… but please, don't unless you have a real reason.
```

If it's confusing, you can visually imagine it more like this: (note that the parenthesis are not valid syntax)

```
        (int*) * c; // A pointer to a pointer to an int
        ((int*)*) * d; // A pointer to a pointer to a pointer to an int
```

Pointer Operators

There are two operators that are especially associated to pointers and pointing:

- the reference operator : &
- the dereference operator : * (this is not the same as when used in declaration, hence the blue)

It is important to remember that while they only apply to pointers, at their core, & and * are just operators, which means the regular rules of operators still apply:

- You can attempt to use them on any value; they may or may not compile. You must be cautious about how you use them.
- They follow the rules of precedences as outlined in the precedence table.
- They return a value when used on an object.

The Reference Operator: **&**

Used to determine where in memory a variable is stored.

```
int num_bags = 500;
int * pointer = &num_bags;
```

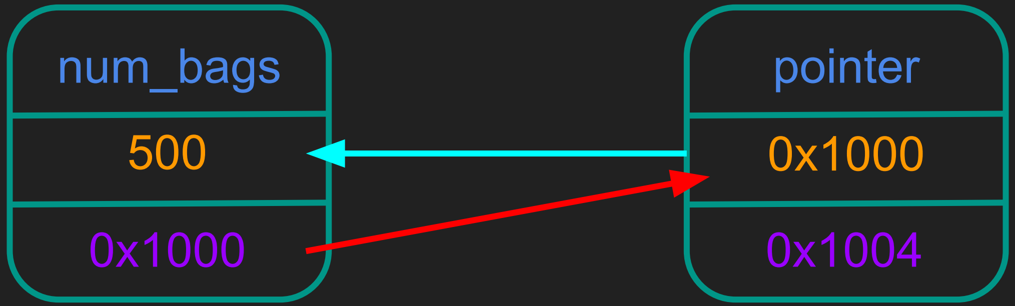
Ask yourself, where is ___?
Ex.
Q: “&num_bags” → Where is num_bags?
A: 0x1000

The Dereference Operator: *

Grabs the value stored in the memory address listed by the pointer.

```
int * pointer = 0x1000;
printf(“%d\n”, *pointer)   //prints 500
```

Ask yourself, who’s in ___?
Ex.
Q: “*pointer” → Who’s in pointer?
A: pointer is 0x1000, which num_bags, or 500.



Pointer Math

Because pointers always assume that they are pointing to an object of a certain type, when pointers increment or get move, they do so by the width of their type; this is called the stride of the pointer.

Ex.

```
int* a        = 0x1000; // stride is 4, because an int is 4 bytes (in this class an int is always 4)
double* b = 0x1000; // stride is 8, because a double is 8 bytes
char* c     = 0x1000; // stride is 1, because a char is 1 byte
a++;
b++;
c++;
printf(“%x”,a); // prints out 0x1004, because the stride of a is 4.
printf(“%x”,b); // prints out 0x1008, because the stride of b is 8.
printf(“%x”,c); // prints out 0x1001, because the stride of c is 1.
```