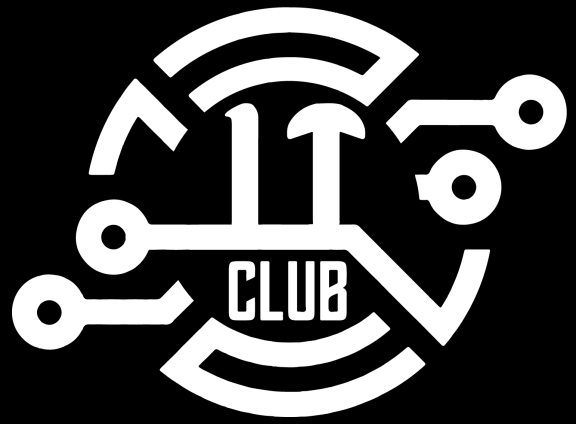
A WORKSHOP ON

THE C PROGRAMMING LANGUAGE



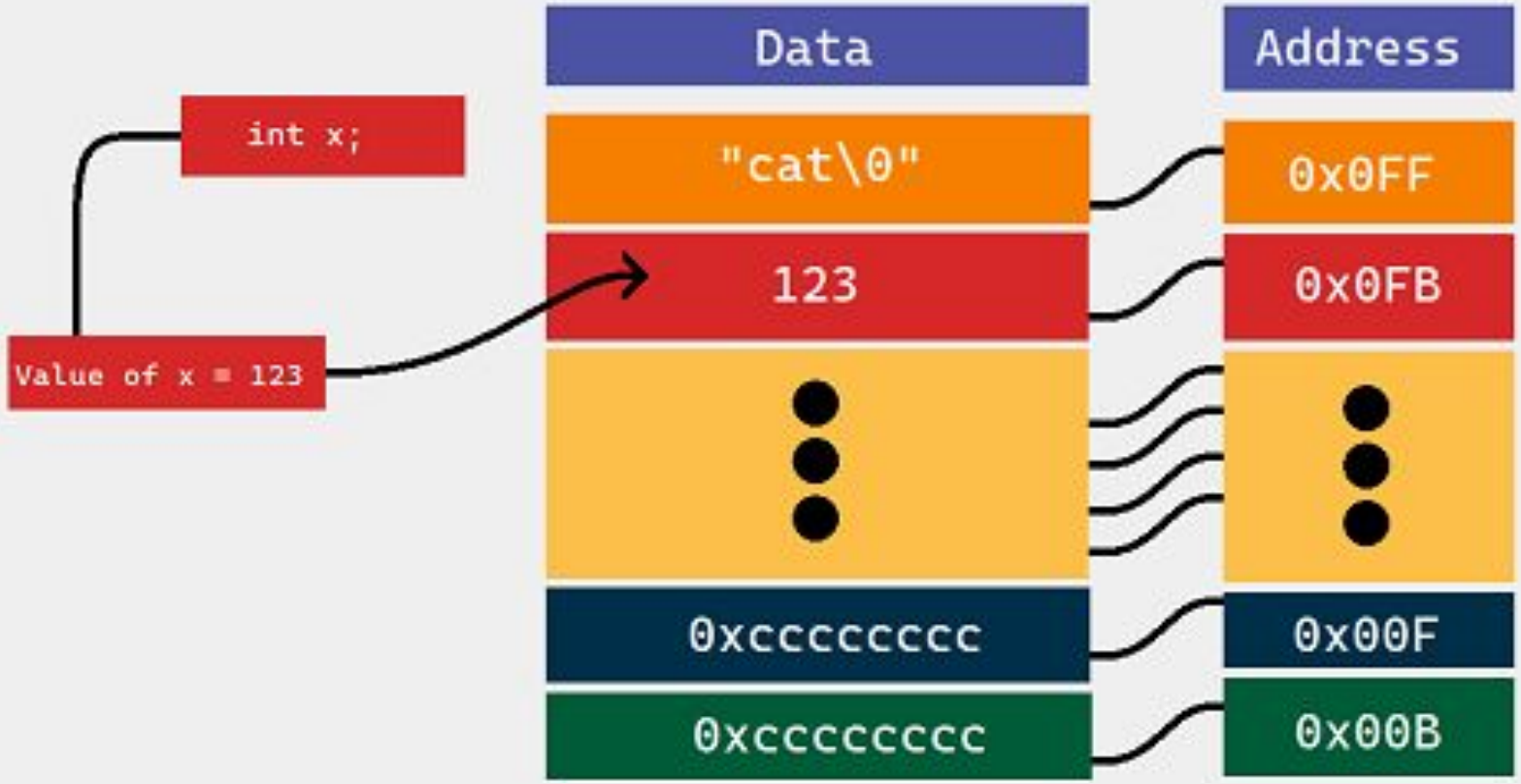
LECTURE 4

The Wizard Points the Way

The big question.

What are pointers?

Pointers are just like any other normal variables, but instead of storing data given by a user, they store memory addresses.



So, what time is it?

That's right, syntax time:

```
int number = 25;
int * pointer = &number;
//the & is the address of operator
int num2_electric_boogaloo = *pointer;
//the * is the dereference operators
*pointer = 60; //changes number as well
```

So, why pointers?

- Mutate functional parameters i.e. pass by address.
- Helps to avoid passing around huge data by copying.
- Arrays decay into pointers when passed into a function.

Pointers hidden in plain sight, Arrays

Believe it or not, arrays too are pointers. Since the basename of the array contains the address of the first element of the array. So we can directly assign it to a pointer.

```
int array[] = {1, 2, 3, 4, 5};
int * ptr_array = array;
ptr_array[1] = 0;
//same as array[1] = 0;
```

So what's the difference:

There's two major differences between arrays and pointers. Firstly the sizeof operator.

```
int array[] = {1, 2, 3, 4, 5};
int * ptr_array = array;
sizeof(array)       //20 (in bytes) since 4 bytes * 5 - 20
sizeof(ptr_array)   //8 in x64 and 4 in x86 based architecture
```

So what's the difference 2: electric boogaloo

Secondly there's a slight difference with the assignment of arrays and pointers.

```
int array2[] = {1,2,3};
array = array2;          //Illegal, can't do this
ptr_array = array2;      //Legal
```

Quick mafs time

Another topic that might seem daunting at first is pointer arithmetic. But once you get used to it, you'll realise there was no need to worry.
The four basic rules:

- Integer addition to a pointer is allowed.
- Taking the difference between two pointers of the same type is allowed.
- Comparison for equality is allowed.
- using sizeof on a pointer is allowed

Why was pointer arithmetic required though?

For this, we'll be coming back to arrays and how they're similar to pointers. Like it was stated, arrays are basically pointers. And they in fact contained the address to the first element. So,

```
char array[] = {'a', 'b', 'c', '\0'};
// '\0' is a null byte character
*array = 'E';              /is valid
*(array + 1) = 'E';
//assigns the second element to zero
```

But wait, what about different data types that take up different amount memory?

Well yes, glad you caught that, as expected from a participant in our event (wink wink). And for that my friend, the compiler comes to the rescue. The compiler secretly multiplies the number we use to add by a sizeof(type pointed to by the pointer). So essentially,

```
int num[3] = {1,2,3};
int * a = num;
a = a + 1
//the compiler basically does a = a + 1 * sizeof(int)
```

Operation Arrow ->

The '.' operator was used to access struct members. But what about pointers to structs? Well for that we got the '->' operator.

```
typedef struct {
   int x,y;
} Vec2;

Vec2 vec2 = {1, 2};
Vec2 * p_vec2 = &vec2;
p_vec2->x = 3;
```

Big brain time

char* (aka c-strings)

Firstly let's get over with char arrays. The string that you might've been familiar with in other programming languages works a bit differently here in C.

```
char str[] = {'a', 'b', 'c', '\0'};
char str2[] = "abc";
//Same thing but str's declaration is a bit cumbersome.
```

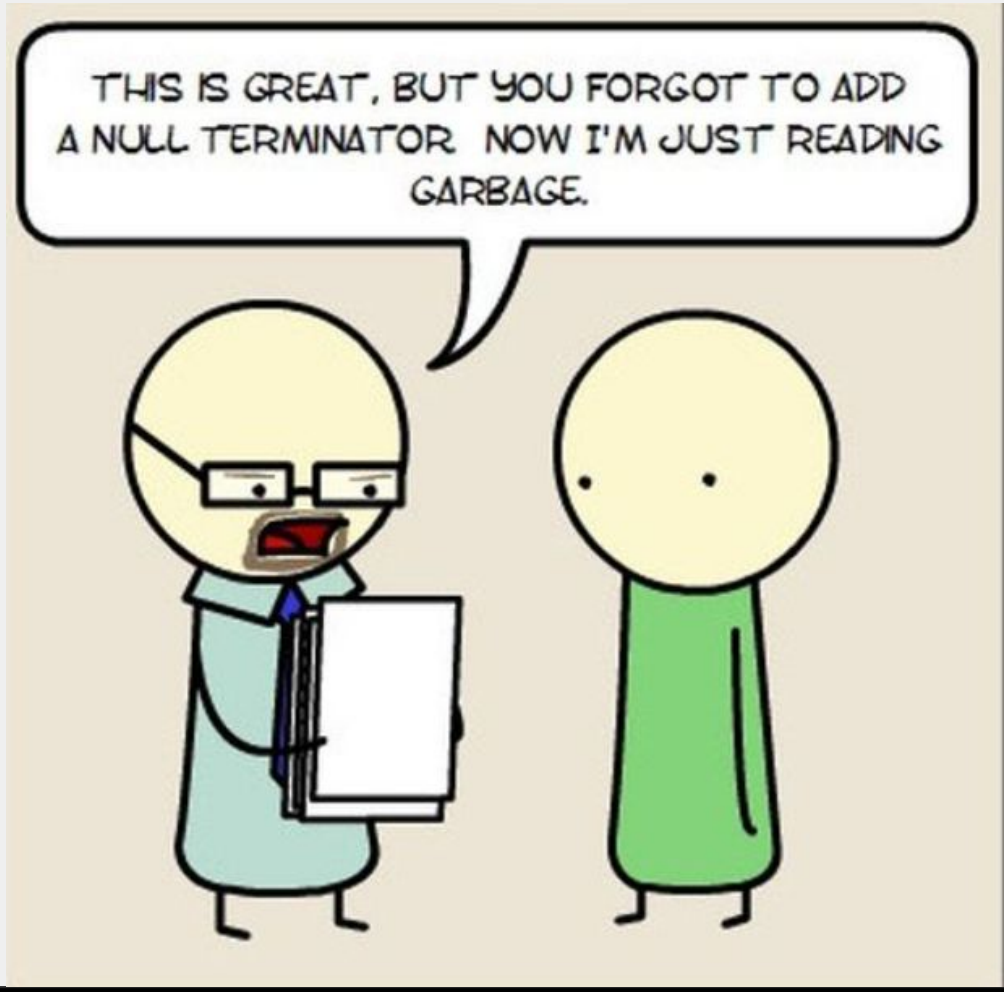
So, what the null?

The ‘\0’ is called the null byte. It is basically a whole byte of memory filled with 0s.

Instead of writing ‘\0’, you can just do 0x00 with is a hexadecimal shorthand for the same thing.

The whole thing with c-strings is that it works off of the null byte instead of keeping the length of the string.

So when you encounter a null byte you’ll realise that you’ve reached the end of the string.



But where’s the char*

We’re getting there, don’t worry. Like we discussed, pointers and arrays are basically the same thing. But there were a few subtle differences. Likewise char[] and char* are different in a slight regard as well.

```
char str[] = “abc”;
char * ptr = “abc”;
//Basically the same thing but,
str[0] = ‘c’; //Allowed, changes str to “cbc”
ptr[0] = ‘c’; //Illegal can’t do this
```

But what's the char*

A c string is an array of characters. Like we discussed an array is a contiguous block of memory. Who's name yields the address of the first element. So dereferencing it gives the first char stored in it. Increasing by one and dereferencing gets you the second character and so on till you've reached the null byte, After that it's undefined behaviour as you'll be touching memory you might not have access to.

```
char* city = "Kathmandu";
//city actually stores the address of the first char 'K'
char ch = *city; //Dereferencing gives a 'K'
ch = *(city + 1); //Dereferencing gives an 'a'
```

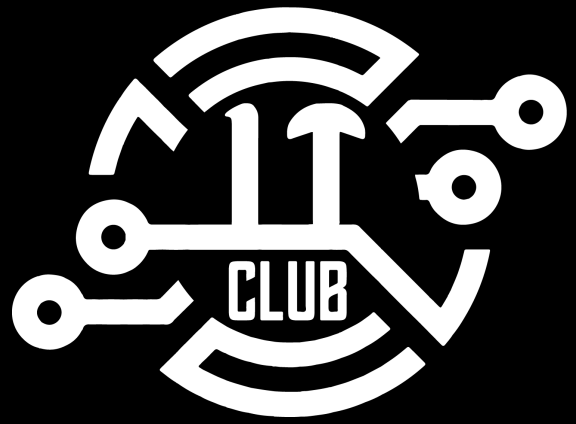
Memory Layout

Since we're already talking about pointers, this would be a great time to talk about the memory layout of a C program. It can basically be divided into four types:

- Stack (Grows Downwards)
- Heap (Grows Upwards)
- Globals/Static
- Code section

A WORKSHOP ON

THE C PROGRAMMING LANGUAGE



LECTURE 4: THE END?

The Wizard Points the Way

Of course not

We’ve barely scratched the surface of pointer power. We covered pointers, but then there’s pointers to pointers, and pointers to those pointers and pointers to...those pointers, wait what. How many pointers are we talking about again?