Finally, the devil is upon us. It's time we discuss a topic which has been put off for far too long.

IMPORTANT

> Before we even start though, Pointers are super complex. So much so that they have a dedicated book of their own called Understanding Pointers in C by Yashwant Kanetkar
>
> ![[Pasted image 20230809211930.png]]
>
> With that said, I'll try my best to encompass most of the day to day relevant stuff about pointers.

We have encountered pointers before, while studying `printf`, `scanf`, and data types but we left at the note "We will discuss them later."

Well now is later.

So let's first begin with the fundamental question:

> What are pointers?

Let's start by learning a bit about RAM.

RAM, or [R]andom [A]ccess [M]emory in essence is a long strip of cells which either have a high voltage or a low voltage

Each of these "cells" is what we call as a "bit".

When you store something on RAM, it is literally stored in a series of these cells the length of which is decided by the type of data. For example, a 32-bit integer `int32_t` is 32 bits long continuous series of these cells, and a 64-bit integer `int64_t` is 64 bits long continuous series of these cells.

But now a question arises as to how on earth does the computer know, what bits comprise of the data we're looking for?

The solution that computer architects came up with was the simplest. Combine these cells in clusters of 8 (a byte) and assign each cluster a linearly increasing address, so let's say I am on the byte with address `add`, the byte directly to the right of this byte will have address 1 more than this byte i.e. `add + 1` and the one to it's left will have address 1 less than this byte i.e. `add - 1`, while the leftmost byte will have the address 0.

NOTE

> There is more to it, but for the sake of simplicity, we will ignore it since it doesn't add to your understanding of what pointers are.

Now, to access the data at the RAM address say `add`, you would go to the ram address of `add` and select <size of data> bytes from there.

But how do you tell computer that this number is an address? Remember all the data inside a computer is really bits even these addresses.

In assembly, you used a special notation to state that this register holds an address, you don't have variables in assembly, you have registers. So say you would put the name of the register inside [] so to use the value at the address stored in the register.

So say you wanted to move the value at the reference which is stored in the register EAX to the register say ESI. You would write the following instruction.

```
mov esi [eax]                ;moves the value at the address in eax to esi
```

Pretty neat right? But as the keen eyed amongst you may have already noticed, there is no indication of how long the data is. So how did computer know that? The answer: it doesn't. It just starts copying bits at the reference in `eax` to `esi` until, `esi` is full.

> So... what if you want to copy 16 bits, and your register is 64 bits?

You can do one of 2 things, copy all of the data and then filter 16 bits with a bitwise and, how will be discussed later in bit operations (I know what you're thinking, it's not that complicated, just a lot less necessary).

Or, you can start copying by skipping the first 8 - 2 bytes i.e. 6 bytes in `esi`.

There is one problem with this approach of handling pointers though, which kills the portability of programs. As discussed earlier, different types of C can have different sizes on different CPU architectures, and it's not really possible to always remember the size of each type on each architecture.

Not to mention, with user declared types like structs, we don't set the memory alignment, C does it for us. In other words, we don't even know how big the type is, we only know what it contains (will be discussed in more detail when discussing structs, don't worry).

So, C decided to make various pointer types instead of just one which Assembly has. Basically, instead of just an address on the ram, you also store what the type of the variable / size of the variable is, this way you know already how much memory is of use when you're using some variable.

IMPORTANT

In C, you CAN duplicate Assembly's behavior by using something called a void pointer. Note the keyword here is can, not should. You should only use void pointer if you are a C veteran and you know what you're doing, because, it just makes your programs orders of magnitude harder to reason with, if you don't know the hardware details.

The syntax for declaring a pointer to a memory which stores variable of type T is

```
T *ptr = &my_var;
```

Here `&` is the reference of operator, it get the address of the variable `my_var`.

Note that, the type of variable is `T *`, i.e. the `*` is the part of type not of the variable, but its written like that, because if you were to declare 2 variables on same line `T* ptr1, should_be_ptr_too;` like so, the type of `ptr1` will be `T *` however that of `should_be_ptr_too` would be T. Why? No one but Dennis Ritchie knows.

Let's write a simple function with pointers to switch the contents of 2 variables, so as to better understand what's going on.

Problem statement:

Write a function in C, which takes in 2 integer pointers and swaps their contents.

```c
#include <stdio.h>
void swap(int *num1, int *num2);

int main(int argc, char **argv) {
    int n1 = 32, n2 = 56;
    printf("n1 = %d, n2 = %d\n", n1, n2);
    swap(&n1, &n2);
    printf("n1 = %d, n2 = %d\n", n1, n2);
    return 0;
}

void swap(int *num1, int *num2) {
    int temp = *num1;
    *num1 = *num2;
    *num2 = temp;
}
```

Output:

```
n1 = 32, n2 = 56
n1 = 56, n2 = 32
```

Note: in swap we use the asterisk (*) while assigning num1 to temp. The * is called the value of operator or the de-refrence operator. It's basically the opposite of the & operator.

So... what is happening in this program? Well let's do a step by step walkthrough of the program.

```
Main starts:
    n1 = 32
    n2 = 56
```

```
print "n1 = 32, n2 = 56"
swap starts:
    num1 = &n1
    num2 = &n2
    temp = *num1 which is equivalent of temp = n1
    *num1 = *num2 which is equivalent of n1 = n2
    *num2 = temp which is equivalent of n2 = temp
```

Learning Tip

You can read `int *ptr = &my_var` as

the 'integer' at 'value of' 'ptr' = the one at the 'address of' of 'my_var'

[[Pointers in C(II)- Arrays & Pointer Arithmetic]]