C LANGUAGE

A basic introduction

POINTERS

WHAT IS A POINTER ?

01

Pointers are **variables** that store **addresses**

Addresses are **memory locations**.

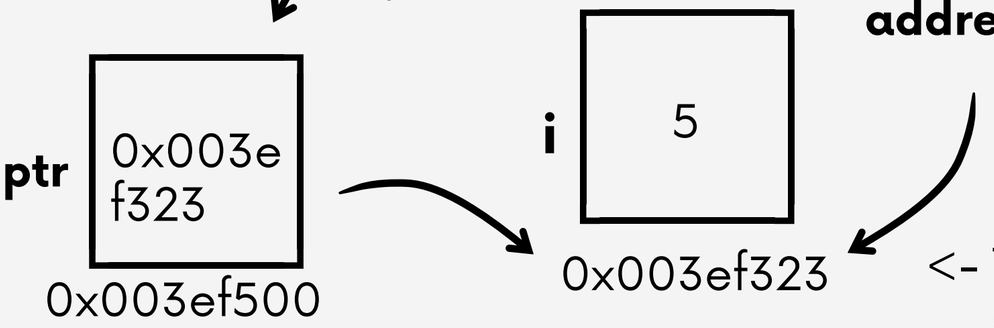
Memory locations are where our program stores data.

```c
#include <stdio.h>

int main()
{
    int i = 5;

    int *ptr = &i;
    return 0;
}
```

Here,
int i is a variable with a value of 5.

This value is stored at somewhere in our **RAM** (Memory)

To access it we need an **address**.



This is a pointer -> **ptr**

look how it's pointing to address of int i;

<- This is a memory location or simply a address

POINTERS

WHAT IS A POINTER ?

01

&(ampersand) is called **The Address Operator .**

It returns a address of a variable.

```
#include <stdio.h>

int main()
{
    int i = 5;
    int *ptr = &i;
    return 0;
}
```

Notice **&i**,
the address is stored in
int *ptr;

A pointer is declared in the following way :

int ***pointer_name** = &**x**;

notice * before **pointer_name.**
This declares that **pointer_name** is a pointer.

And we are assigning the **Address** of **x variable** into **pointer_name** which is also a **variable**.

POINTERS

WHAT IS A POINTER ?

01

We can also do it in the following way:

```c
#include <stdio.h>

int main()
{
    int i = 5;
    int *ptr;
    ptr = &i;
    return 0;
}
```

it's essentailly the same thing

A pointer can also be declared in the following way :

```c
int *pointer_name;
```

for multi pointer :

```c
int *ptr1, *ptr2;
```

we can assign values to them later on.

POINTERS

WHAT IS A POINTER ?

01

```c
#include <stdio.h>

int main()
{
    int i = 5;
    int *ptr;
    ptr = &i;
    printf("Address of i : %x\n", &i);
    printf("Address in ptr : %x\n", ptr);
    printf("Address of ptr : %x\n", &ptr);
    return 0;
}
```

Output :

```
Address of i : 989f409c
Address in ptr : 989f409c
Address of ptr : 989f40a0
```

Note : The addresses are random, they are not supposed to be same all the time. They reset after each program ends.

This shows a proof how the **pointer** and what's it's **pointing to** works.

POINTERS

MORE ABOUT POINTERS:

01

- A pointer holds an address
  - For 32-bit systems **sizeof**() ANY pointer is 4 bytes
  - For 64-bit systems **sizeof**() ANY pointer is 8 bytes

- A pointer variable can be pre-fixed or post-fixed with increment or decrement operators.

- A pointer can point another pointer too! Here, p2 is pointing to a pointer.

```
int x=5, *p1, **p2;
p2 = &p1;
```

POINTERS

DEREFERENCE OPERATOR

01

A **dereference** is simply using * before a **pointer** to access the **elements** it pointing to.

the * is called the **dereference operator**

```
#include <stdio.h>

int main()
{
    int i = 5;
    int *ptr = &i;
    *ptr = 6; // i = 6 now
    return 0;
}
```

Notice the key difference,

declaration of a pointer needs a * too. It points to any variable.

but in order to access that variable, let's say modify the value

we need to **dereference** it putting * before the pointer and assing a new value

POINTERS

NULL POINTERS

01

The Null Pointer is the pointer that does not point to any location but NULL.

```
int *ptr = NULL;
```

Now this pointer is pointing nothing!!!
This is great for initialization.

POINTERS

VOID POINTERS

01

Pointers is simply a variable containing a address.

Let's say we declared it and don't want to just associate it to a address of certain data types.

That's where void pointers comes in.

The void pointer in C is a pointer that is not associated with any data types

```
void *ptr;
```

Now it can hold any types of data's addresses. Be it float, int, char etc.

POINTERS

VOID POINTERS

01

But void pointers cannot be **dereferenced.**

```
#include <stdio.h>

int main()
{
    int i = 5;
    void *ptr;
    ptr = &i;
    printf("%d\n", *ptr);
    return 0;
}
```

Output :
Compiler Error: 'void*' is not a pointer-to-object type

They are type-casted.

```
#include <stdio.h>

int main()
{
    int i = 5;
    void *ptr;
    ptr = &i;
    printf("%d\n", *(int*)ptr);
    return 0;
}
```

Output :
5

POINTERS

POINTERS ARITHMETIC

01

Pointer arithmetic are some basic operations we can do on the pointers.

This effects in changes of the **memory address** the pointer is holding.

It would shift to a different **memory address** in the process.

POINTERS

POINTERS ARITHMETIC - INCREMENT/DECREMENT

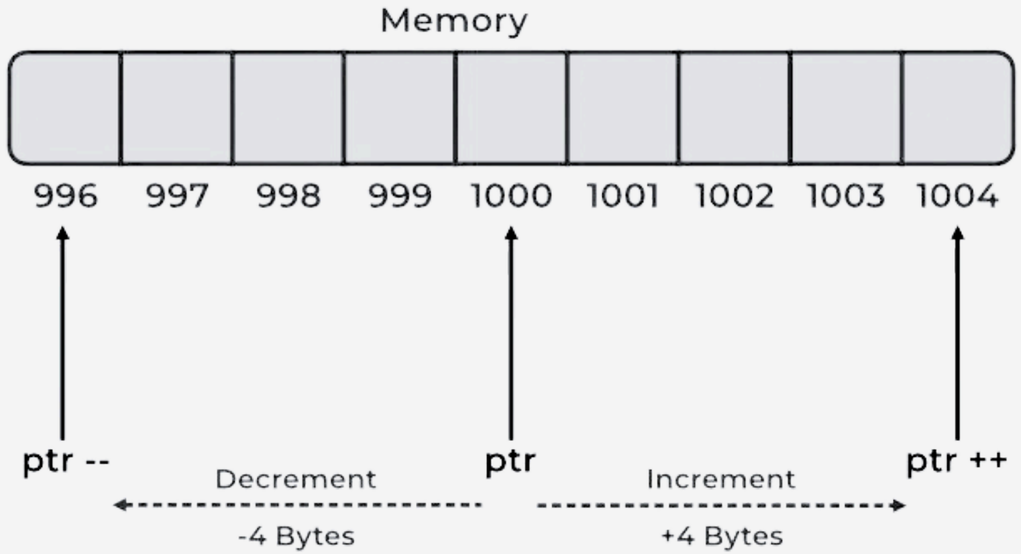
01

```c
#include <stdio.h>

int main()
{
    int i = 5;
    int *ptr = &i;
    printf("%d\n", ptr);
    ptr++;
    printf("%d\n", ptr);
    ptr--;
    printf("%d\n", ptr);
    return 0;
}
```

Output :

```
875787964
875787968
875787964
```

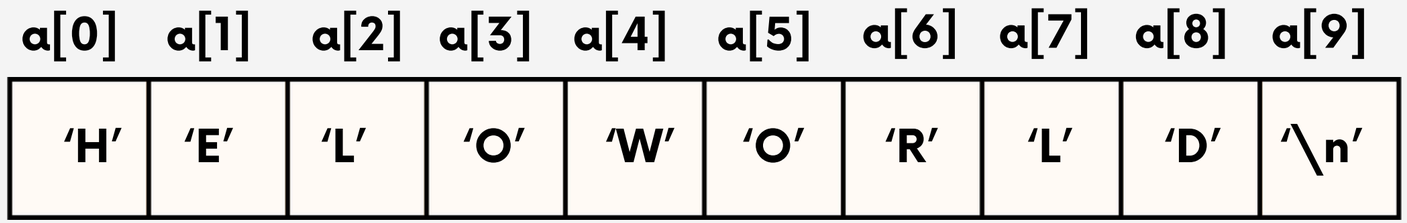
Notice the difference,
There's a jump of 4bytes
in the memory location.



POINTERS

ARRAY AND POINTERS

01




```c
#include <stdio.h>

int main()
{
    char a[] = "HELOWORLD";
    char *ptr;
    ptr = a;
    printf("%c\n", *ptr);
    return 0;
}
```

Output :

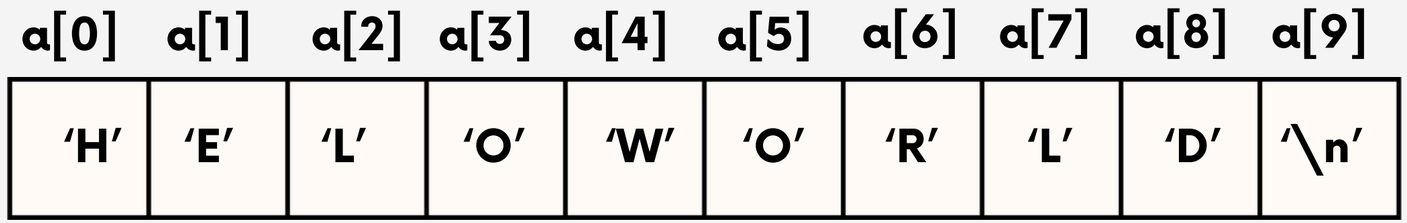
H

By default,
pointers takes the addresses of the first element of an array.

POINTERS

ARRAY AND POINTERS

01




```c
#include <stdio.h>

int main()
{
    char a[] = "HELOWORLD";
    char *ptr;
    ptr = &a[4];
    printf("%c\n", *ptr);
    return 0;
}
```

Output :

W

Now we assigned the address of a[4] in the pointer

POINTERS

ARRAY AND POINTERS

01

```c
#include <stdio.h>

int main()
{
    int x[5] = {1, 2, 3, 4, 5};
    int* ptr;
    ptr = x;
    printf("*ptr = %d \n", *ptr);
    printf("*(ptr+1) = %d \n", *(ptr+1));
    printf("*(ptr-1) = %d", *(ptr-1));

    return 0;
}
```

Output :

```
*ptr = 1
*(ptr+1) = 2
*(ptr-1) = 32767
```

pointers of an array by default has the **0th index's address** as their stored data.
in this case the **x[0]**

so *ptr = 1
if we add ***(ptr + 1)** , we go to the next index which is **x[1]**

so *ptr = 2

x[0]	x[1]	x[2]	x[3]	x[4]
1	2	3	4	5

POINTERS

ARRAY AND POINTERS

01

```c
#include <stdio.h>

int main()
{
    int x[5] = {1, 2, 3, 4, 5};
    int* ptr;
    ptr = x;
    printf("*ptr = %d \n", *ptr);
    printf("*(ptr+1) = %d \n", *(ptr+1));
    printf("*(ptr-1) = %d", *(ptr-1));

    return 0;
}
```

Output :

```
*ptr = 1
*(ptr+1) = 2
*(ptr-1) = 32767
```

But, ***(ptr-1)** is different.

we can generally assume we're doing index addition.
&x[0 + 1] = x[1]
but, **x[0-1] = x[-1]**

there's no such thing as **x[-1]** . So we will either get 0 or a random Garbage value.