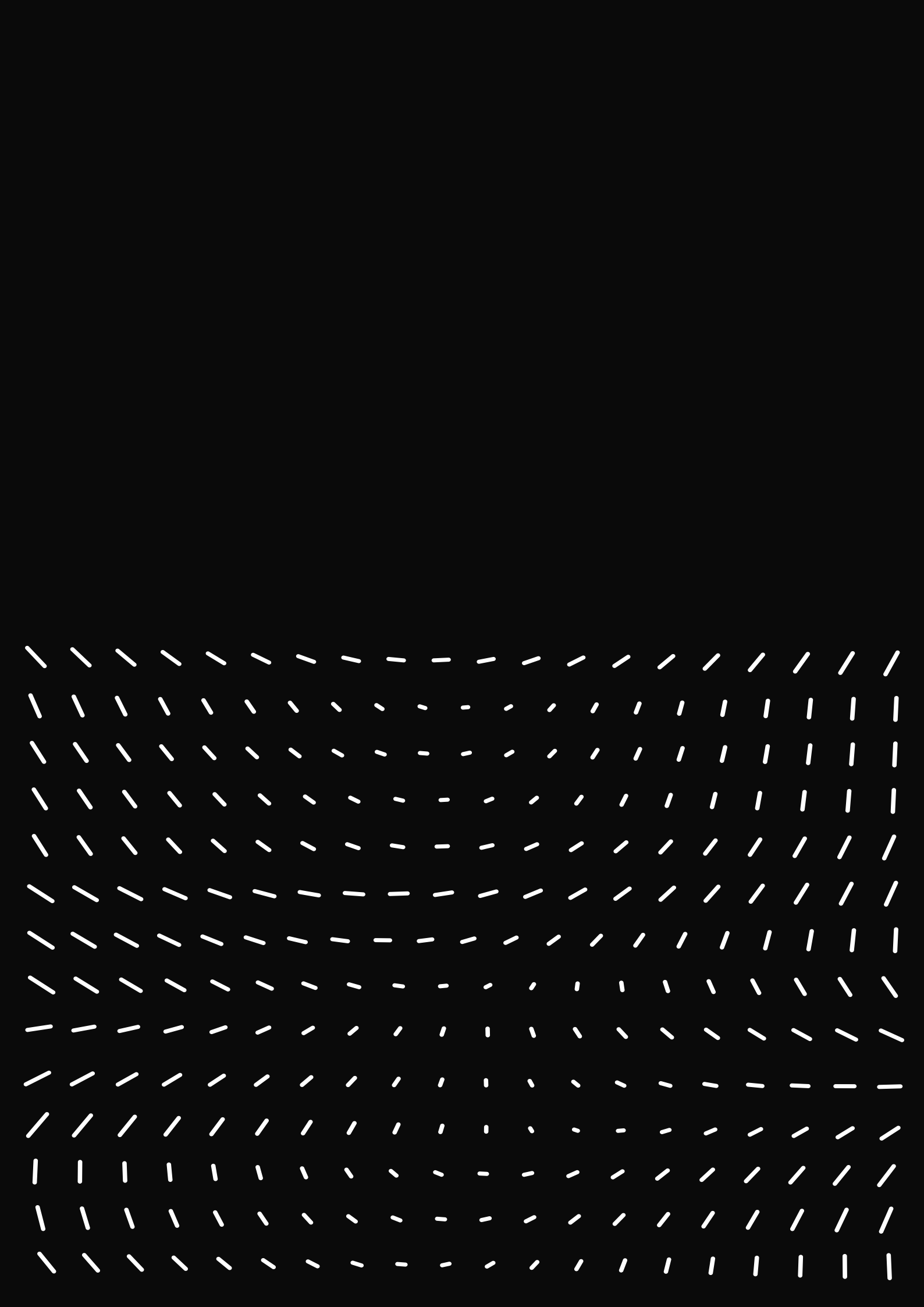
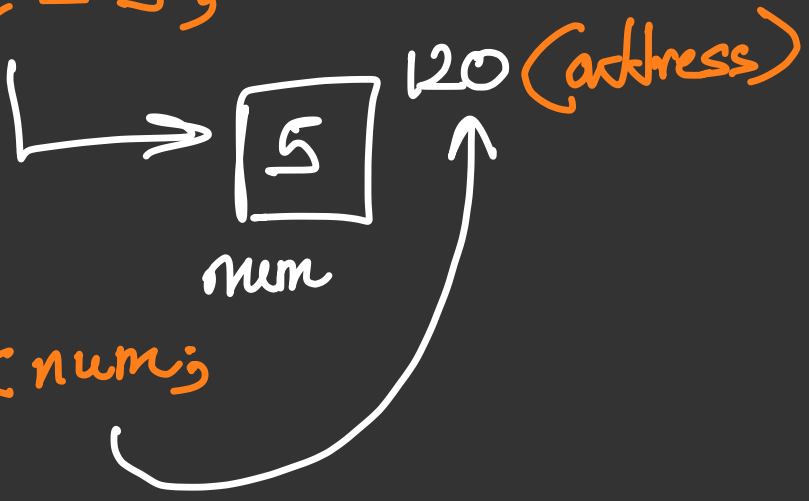


Lecture 25

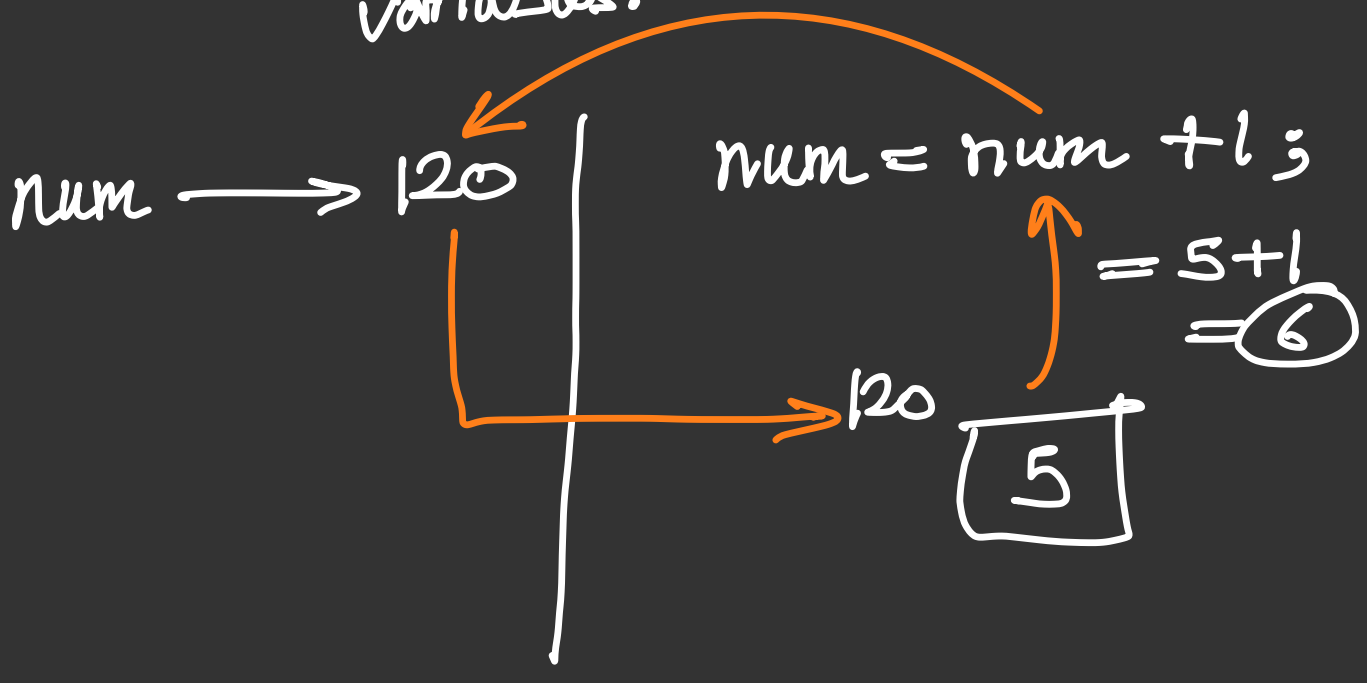
Pointers

int num = 5;



✡ Symbol Table

→ an important data structure created and maintained by the compiler in order to keep track of semantics of variables.



Address of operator

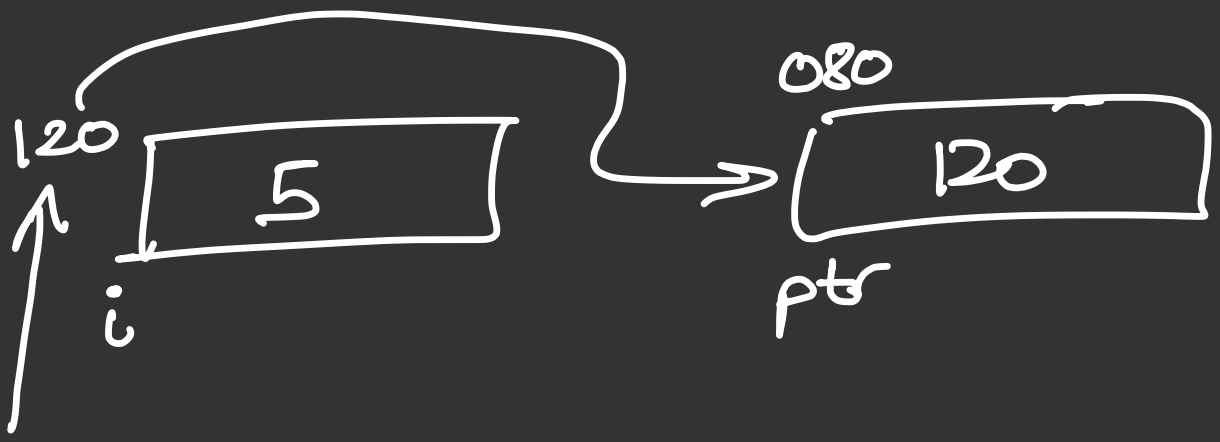
→ &num

= 0x16b11f488

in hexadecimal format

To store the address → POINTER

int *ptr = &i;



(*ptr)++;

* Always initialize the pointer, don't just declare it.

→ pointers.cpp

```
int *p = &num;
```

→ p is a pointer to int

```
char ch;
char *p = &ch;
```

} Datatype needs to be same.

*p → dereference operator

example:


```
int *p = &num;
cout << *p << endl; // 5
cout << p << endl; // 100
```

Pointer always stores address so, size of pointer → 8

* If you are trying to access an address that does not exist → segmentation faults.

```
int *p = 0;
p = &i;
```

NOT `*p = &i;` ✗

* Any changes made to *p will impact value of i also.

Copying a Pointer

```
int *p = &num;
int *q = p;
```


```
p = p + 1;
```

→ p = 104 (next integer address)

Types of Pointers ⟶

① Null Pointer

⟶ pointing to nothing

```
int *ptr = NULL;
int *p = 0;
```

② Double Pointers

```
int a = 10;
int *p = &a;
int **q = &p;

cout << a << *p << **q << endl;
        10    10     10
```

&a = p = *q ∑ address of a

&p = q ∑ address of p

3 Void Pointer

→ has no associated type with it.

```
void *ptr = 0;
```

4 Wild Pointers

→ declared but not initialized.

5 Dangling Pointers

→ function call

→ deallocation of memory

→ Variable goes out of scope.

Example:

```
float f= 10.5;
float p= 2.5;
float *ptr= &f;
(*ptr)++;
*ptr= p;
cout << *ptr << f << p << endl;
```

f=11.5

f=2.5

2.5 2.5 2.5

ptr is still &f

example:

```
int a[] = {1, 2, 3, 4};
cout << *(a) << " "
     << *(a+1);
```

1 2

example:

```
char b[] = "xyz";
char *c = &b[0];
cout << c << endl;
```

entire array is printed.

* b also points to the address of the first character &
when cout<<b entire array gets printed, so similarly.

Lecture 26

```
int arr[] = {1, 2, 3, 4, 5};
```

arr → points towards the address of the first element

Thus,

arr = &arr = &arr[0]

and

arr[0] = *arr = 1

*(arr+1) = 2

⇒ arr = address

address +1 → in integer ie going to the next integer

*(address) = value at address.

→ pointers_in_array.cpp

arr[i];

↳ *(arr+i); } important

AND

⟹ arr[i] = i[arr];

because in logic *(arr+i) it doesn't matter which is first.

Difference ⟶

Array ⟶ int arr[10];

Pointer ⟶ int *p = &arr[0];

(1) sizeof(arr) ⟶ 40 = 10×4

sizeof(p) ⟶ 8

came out to be 4 in my laptop

(2) &p ≠ &arr

as, &arr = address of arr[0]
= p

and,
arr[0] = *p

bcuz,
&p → points to the address of the pointer p in the memory.

(3) Symbol table ka content cannot be changed.

arr = arr + 1; → ERROR

p = p + 1;
↳ *p = arr[0+1];

As p = &arr;
nothing to do with &p.
(address of pointer p)

int *ptr = arr;

*ptr = arr[0]; (the value)

&ptr = address of pointer P

ptr = address of arr[0]

Char Array

→ charArray_pointer.cpp

char ch[6] = "abcde";

↑ 5+1 for null character

cout << ch;

↳ prints "abcde" NOT The ADDRESS

char c = &ch[0];

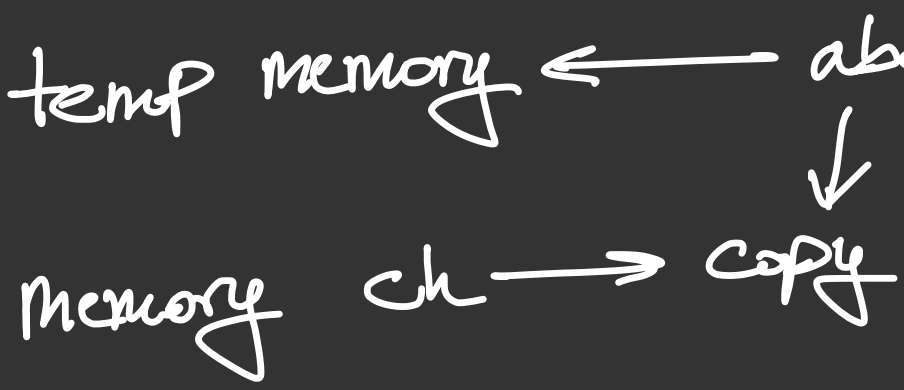
cout << c << endl;

↳ prints "abcde" NOT THE ADDRESS.

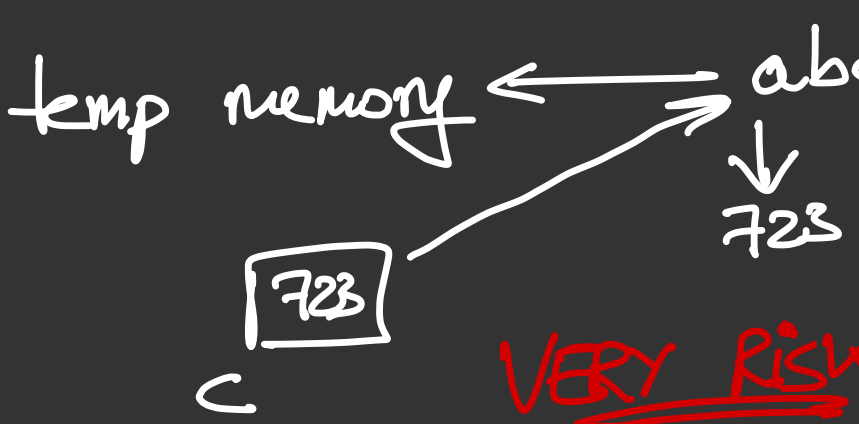
* Printing only stops in char array when we reach null character '\0'

Behind the scenes →

1. char ch[6] = "abcde";



2. char *c = "abcde"



VERY RISKY as its pointing to address in temporary memory

Functions

→ functions_pointer.cpp

✡ If you are passing a pointer in function, you are passing an address, so `p = p + 1;` won't make any difference to the pointer in main, but `*p = *p + 1;` will change the value everywhere because it is changing the value at the address.

✡ When we pass array in functions, only the address of the first element as pointer is passed.

Thus, to include size of array as one of the parameters is also important.

☆ We can send part of array also in the parameters, by passing the address of the starting element.

IMPORTANT USECASE →

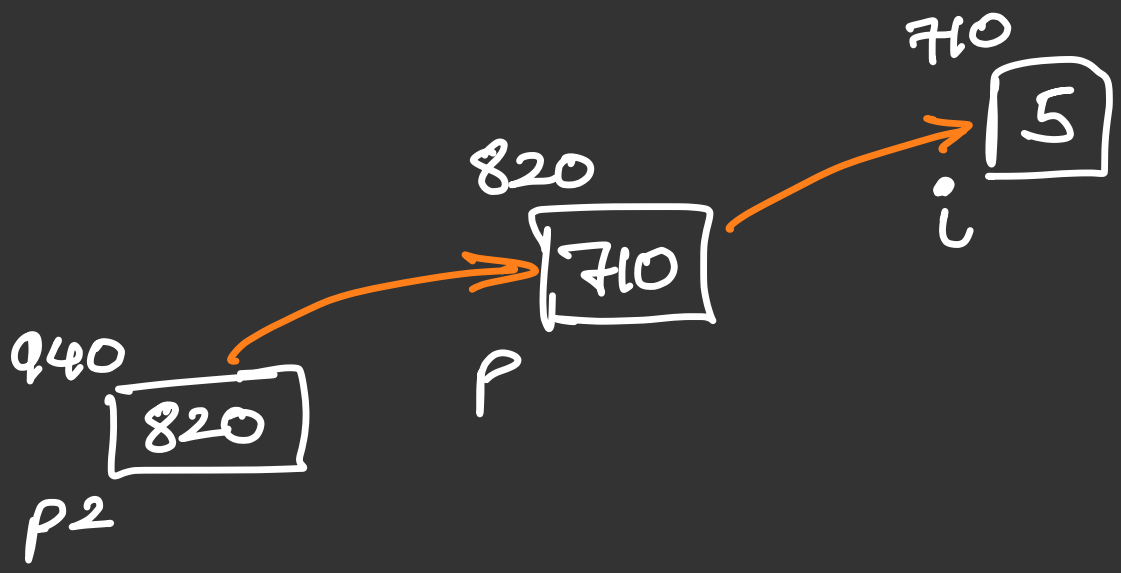
by just (arr+i)

Lecture 27

Double Pointer

```
int i = 5;
int *p = &i;
int **p2 = &p;
```

2 → double pointer




```
**p2 = *p = i;
```

↳ star refer to the depth

Passing ***p in a function

```
update (int **p2)
{
    p2 = p2 + 1;
```

→ it updates for the local pointer

```
    *p2 = *p2 + 1;
```

→ updates the address stored in p by +4. (coz ints)

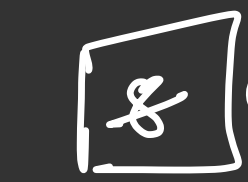
```
    **p2 = **p2 + 1;
```

→ change value of i from 5 to 6.

```
}
```

example:

```
int a = 8;
int *p = &a;
cout << (*p)++ << endl;
```



⇒ but because post increments, 8 will be printed

example:

```
int first = 2;
int *p = 0;
p = &first;
*p = first;
```

✓

✗ Wrong → ERROR

~~*~~ MCQ's in the end of the lecture. (from 31:00)

Good for PRACTICE

Lecture 28

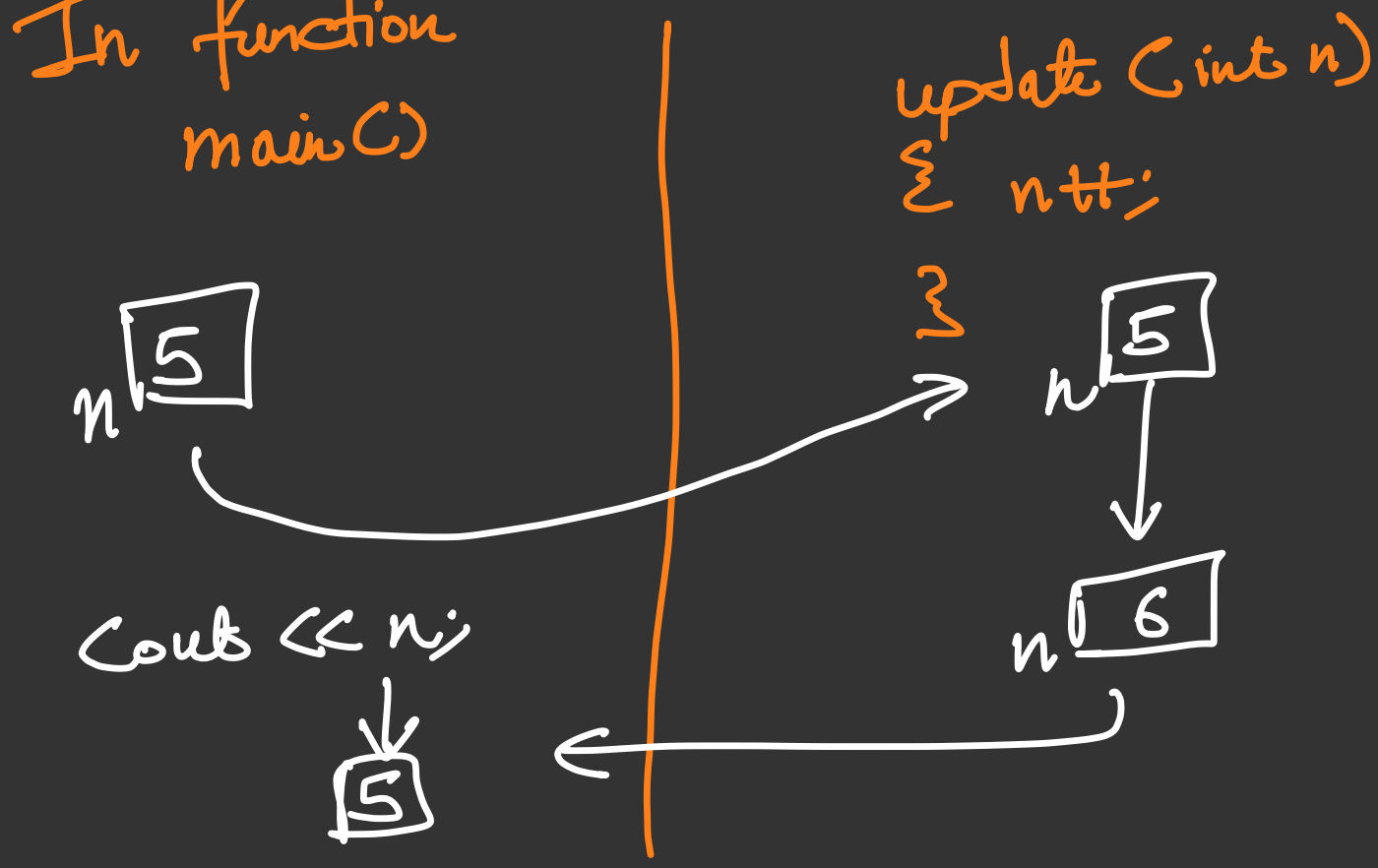
Reference Variable

→ same memory but different names.

Syntax →

```
int i = 5;
int &j = i;
```

→ reference_variable.cpp



Still 5 gets printed from main, because we PASS BY VALUE, value of variable n gets printed and a copy of n is made and n++ happens for that.

Easy way to understand

→ update (int m)
{
m++;
}

Now m is there instead of n but same function.

Passing reference variable as a parameter ⟶

```
update (int &n)
{
    n++;
}
```

↳ no new copy is made instead n here is referencing to the address of the variable n passed.

ERROR ✓

```
int & update (int n)
{
    n++;
    int &ans = n;
    return ans;
}
```

✗ BAD PRACTICE

Coz ans only has scope within this function so gives error

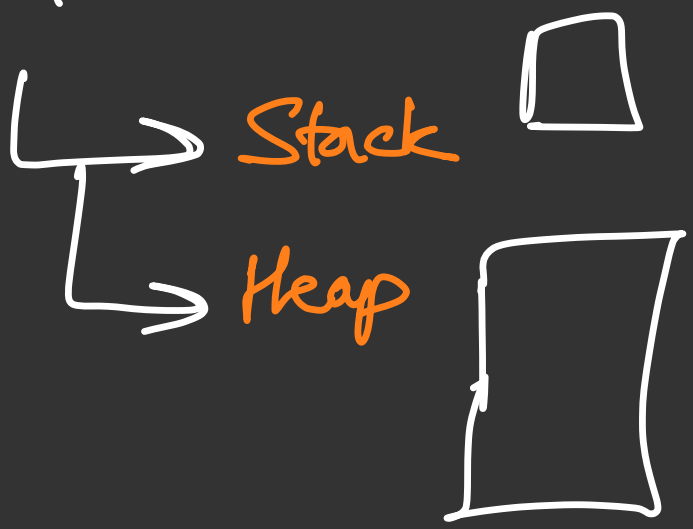
Another BAD PRACTICE →

```
int n;
cin >> n;
int arr[n];
```

→ as we will get value of n at runtime

but the array should be initialised at compile time only.

Program starts



✱ So if the program knows at compile time only that more space will be needed the more stack memory will be

alloted before runtime.

Thus, program crashes if the memory needed is more than stack memory

→ Till now all the memory we used in our program what using Stack Memory.

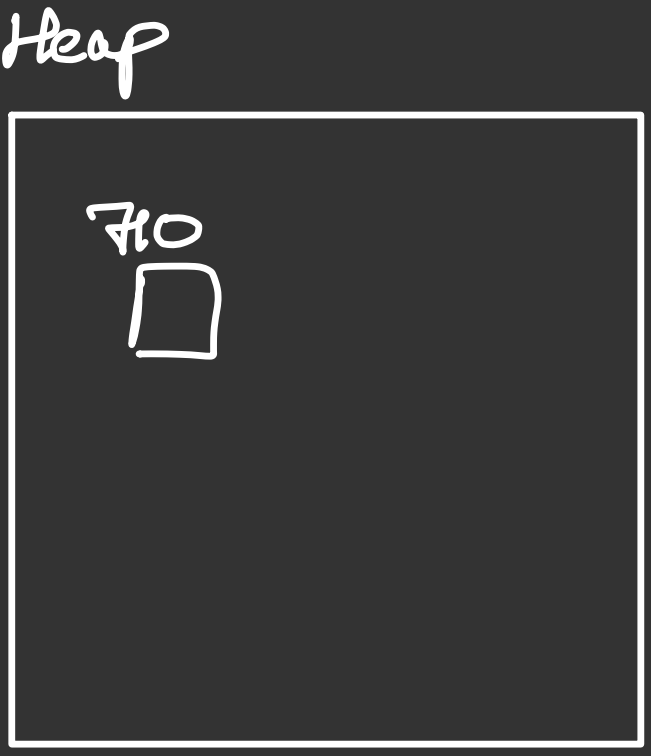
So, if you want to get the array size from the user and then assign the array, we use HEAP memory using 'new' keyword.

Stack → Static memory allocation

Heap → Dynamic memory allocation

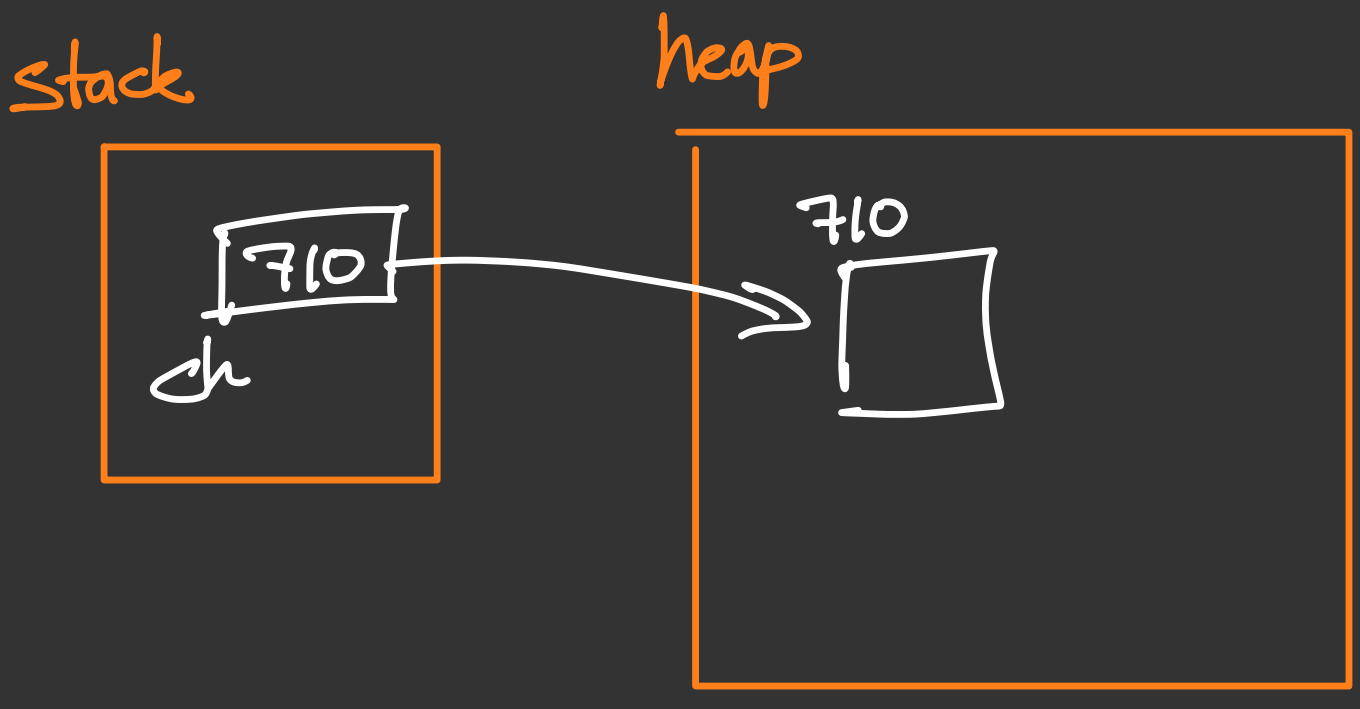
new char;

- returns address
- no variable name



Thus,
to store address we use pointers

char *ch = new char;



* ch is stored in stack

Array in heap →

```
int *arr = new int[5];
```

↳ address of the first element in heap memory

Variable size array →

→ Variable_size_array.cpp

Difference →

Static	Dynamic
① `int arr[50];` ↓ 4×50 = 200 byte	`int *arr = new int[50];` ↓ 8 byte (stack) ↓ 50×4 = 200 byte (heap) ↓ 208 byte

2

```
while (true)
{
  int i=5;
}
```

as soon as it comes out of the bracket the memory gets released.

```
while (true)
{
  int *p= new int;
}
```

p memory in stack gets free but every time new memory will be assigned in heap memory.

Thus, as soon as heap memory is full, the program will crash.

3 Memory automatically gets released.

We have to manually release

↳ 'delete' keyword

delete for dynamic memory allocation

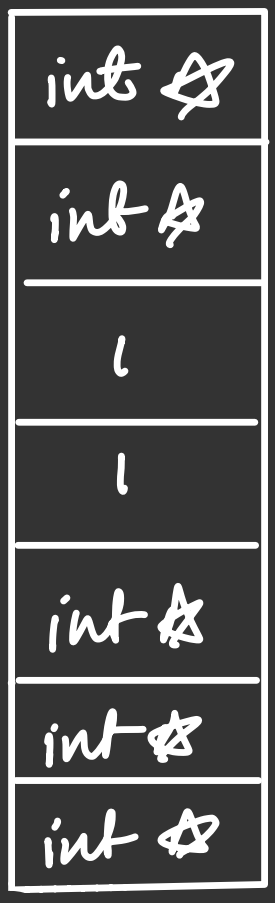
1. `int* i = new int;`
   → `delete i;`
2. `int* arr = new int[50];`
   → `delete [] arr;`
   (for Array)

* HEAP works or allocates memory during Runtime

Lecture 29
2-D Array using dynamic memory allocation

`int ** arr = new int*[n];`

↳ will give an array of pointers



↳ `new int[m];` for each pointer

→ 2dArray-dynamic.cpp

Remember to RELEASE/FREE the memory after usage.

FACT →

→ Local Variables are stored in Stack

→ Static and Global variables are stored in permant storge area.

JAGGERED Array

↳ Dynamic memory

→ basically 2D Array of different row lengths.

→ jaggered_array.cpp

Lecture 30

Macros

→ #define

#include <iostream>

→ preprocessor directive

Macro

→ a peice of code in a program that is replaced by value of macro.

Example:

#define PI 3.14

↓ no semicolon

→ does not requires any memory / storage

→ At the compile time, wherever PI is written in the program, that is replaced by 3.14.

→ macro.cpp

Types of Macros →

① Objects-like

```
#define DATE 31
```

② Chain Macros

```
#define INSTAGRAM FOLLOWERS
#define FOLLOWERS 330
```

③ Multi-line

```
#define ELE 1, \
            2, \
            3

int arr[] = { ELE };
```

④ Function like

```
#define min(a,b) (((a)<(b)) ?
                  (a) : (b))

#define PI 3.14
#define AREA(r) (PI*(r)*(r))
```

⚠ No need of brackets for variables

Global Variable

better way

One way to share variables b/w functions → Reference variable

Other way → global variable

Local Variables

→ they exist between scopes only and cannot be accessed outside particular brackets.

→ global_variable.cpp

⚠ BAD PRACTICE to use them.

→ because it can be changed in any function and that change will reflect anywhere.

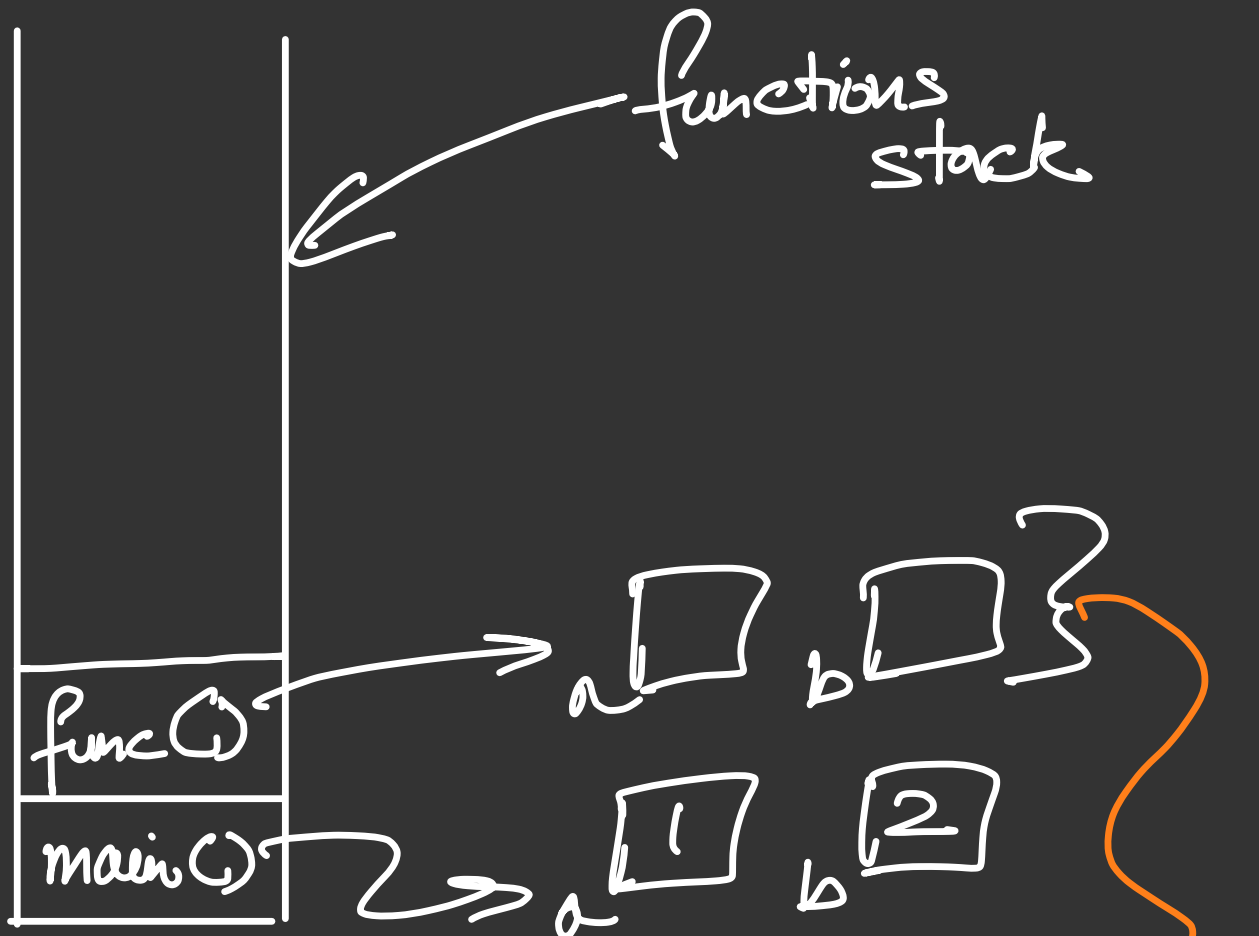
Inline Functions

↳ are used to reduce the function calls overhead.

Code example :—

```
void func(int a, int b)
{
    a++;
    b++;
    cout << a << b << endl;
}
int main()
{
    int a=1, b=2;
    func(a,b);
    return 0;
}
```

What happens ⟶



as value is passed thus copy variables are mark in the memory

↳ PERFORMANCE HIT

↳ function call + memory

Memory can be saved by using reference variable →

```
void func(int &a, int &b)
{
    ==========
}
```

for function call → INLINE FUNCTION

```
inline int getMax(int &a, int &b)
{
    return a>b ? a : b;
}
```

* Only when only 1 Line inside function

→ works same as macro, during compile time wherever getMax is written will be replace with a>b ? a : b

Thus, no extra usage
+
no function call

⟶ inline_functions.cpp

Default Arguments

```
void print2(int arr[], int size,
            int start)
```

✱ We can make start as default argument by assigning some default value, in case value is not passed or only two values passed.

```
void print2(int arr[], int size,
            int start = 0)
```

IMPORTANT

↳ the rightmost argument should be the default one or if more than one the start from RIGHT. ←

→ default_argument.cpp

☆ Did some experiment with pointers

→ experiments.cpp