UNIT-III Syllabus:

Introduction, pointers for inter function communication, arrays of pointers, pointer arithmetic and arrays, passing an array to a function, memory allocation functions, pointers to functions, pointers to pointers.

Strings: Concepts, String Input/output functions, arrays of strings, String manipulation functions.

Pointers in C:

What is pointer?
Need for pointer
How to declare a pointer
How to initialize a pointer

What is pointer?

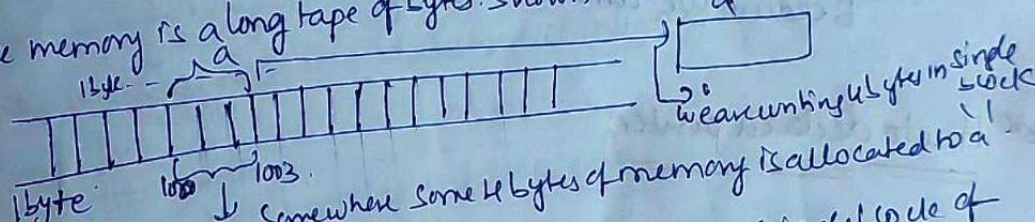
Before discussing about pointer, we will discuss about variable.

We have some fundamental datatypes in C. Like int, float, double, char ...

`int a;` → this is a variable declaration.

If we declare like this: Space will be allocated to the variable and the variable name is 'a' and datatype is int → so it will take 4 bytes.

and like memory is a long tape of bytes. shown like below:



→ Address is what when we declare / show it as single block of memory for a variable →

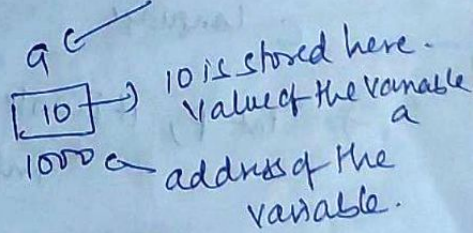
Base address → It means the address of the first byte.

Which will be the address of this variable i.e. (1000 to 1003)

but we take BA as 1000 only.

→ We didn't initialize it / If we don't initialize it it will have some garbage value

Lets initialize as: `int a=10;`



Three things the variable will have: name, value and address.

pointers are also variables or we can call them as special variables and these are not like normal variables

> pointers are special variables which contain address of any other variable.

pointer doesn't contain any simple value like 10, 5 or like integer value or float value.

> pointer always contain address of any other variable.

> pointers are derived datatypes

Because we drive it using these fundamental datatypes.

** How to declare a pointer:

General syntax:

> datatype *(asterik) pointername

This is the general syntax to declare a pointer variable.

→ If i write int *p; int* p, int * p; ⇒ these are same in C. Language.

→ int p; if i write like this p will contain which typed value? integer value.

But if you write int *p; ⇒ i.e. asterik followed by p, it means p is known as a pointer variable. which will contain the address of any other variable.

More specifically:

int *p; → means it will store the address of a integer type of a variable.

Here we cannot say the data type of the pointer is integer.

NO

float *p; → We cannot say the data type of pointer is float.

This is the common mistake what many do.

In int, here pointer contains address and in float *p; also pointer p contains address.

In both the cases pointer will contain address.

How can we say that here data type is int and here data type is float?

Because pointer contains address only

↳ Now what does this data type mean, int, float, double.

If i write double *p;

→ Why we are writing this, int, float and double?

It means the pointer 'p' is containing the address of a variable whose data type is integer.

We can say for suppose 'p' is containing the address of a

Because we can say the data type of a is integer.

In float *p; → Here p will contain the address of a variable whose data type is float.

Suppose I declare:

float b=1; and float * p;

here in this p contains the address of b and b's datatype is float.

Ind double *p; → here p will contain the address of a variable whose datatype is double.

Size of pointer is 4 bytes for 32 bits machine.

How to initialize a pointer:

p=? → Now p will contain the address.

How we will get the address of a variable or any other variable?

We will get the address using address of (&) operator.

int a; int *p; } We declare like this.

and p = &a; → We will get the address like this.

But if you write p = &b; → it is not correct. Why? here.

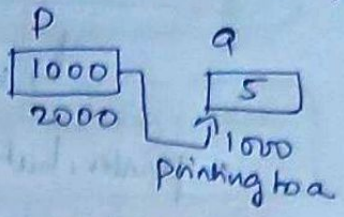
Because according to the declaration of pointer, p should contain the address of the variable whose datatype is integer here and here I am initializing p=&b; and datatype of b is float so it will be illegal.

→ Now if you declare a pointer something like:

int *p;

Then name of pointer is p, and obviously this is a variable or a special variable which contains address, but as it is also a variable, it will also takes some space in the memory, so in the memory, some memory

is allocated to this p, so obviously it will have its own address also
for ex: we will consider address of pointer variable 'p' as 2000,
and address will be in Hexadecimal form.



address of p is 2000
suppose address of a is 1000

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

So in p the value we have is 1000 → which is the address of 'a' variable.

So we can say the 'p' is pointing to 'a'. or we can say the 'p' points to 'a'.

Somewhere we say like "ptr points to a" means "ptr" is a pointer variable which is pointing to variable a.

points to ⇒ means it will contain the address of this a or we can say any variable address which it is pointing to.

Declaration is going to tell the Compiler 3 things!

1. — * (asterik) ⇒ means, the variable which is followed by * is a pointer type variable.
2. — p is having some space in memory (address in memory) may be like 2000
3. → p is having an address of "a variable whose datatype is integer" i.e. it is having address of an integer datatype.

⇒ You can do declaration and initialization in same step!

```
int *p = &a;
```
→ ✓

```
int a=10, *p = &a;
```
✓

```
float x, y;
int a, *ptr;
ptr = &x;   ✗
ptr = &a;   ✓
```

`int *p = &a, a=10;` → wrong.

↳ why so you have declared address of a in pointer, but you have not declared a yet. and you are declaring 'a' after this, so we dont have 'a' variable in memory so `int *p=&a` / any address of a in memory is not available, so it will be illegal.

printing the value of 'a':

If we want to print the value of 'a' variable then:

We have 2 ways:

We can directly print like:

```
printf("%d", a);
```

printing the value of 'a' variable using pointer variable 'p':

Because pointervariable is pointing to a, we can print the value of a using p like below:

How to access the value of 'a' using pointer?

Here we have to use a special operator called as

`*` → it is "dereferencing" operator.

↳ we can call this even as "Indirection" operator

`*` → star.

Before this:

If we declare a pointervariable as:

```
int *ptr;
```

But if we don't initialize the above pointer it will point to some unknown location and an initialized pointer are risky to use.

So, Before using the pointers we have to initialize them in the program

Int *ptr; ptr=&b; ⇒ It will not give error, but it will give wrong output

We need to take care of datatypes while initializing pointers.

Ex: double, float b; float *ptr; ptr=&b; → ✓

& (address of) and * (Indirection / dereferencing) operators:

We use two special operators while we deal with pointers:

1. address of (&) operator
2. Indirection / dereferencing (*) operator.

& → will give address of any operator.

* → Star symbol → It is known as indirection operator, we also call it as dereferencing operator which is very important when we are dealing with pointers.

→ Many get confused while using the "*" operator with pointers.

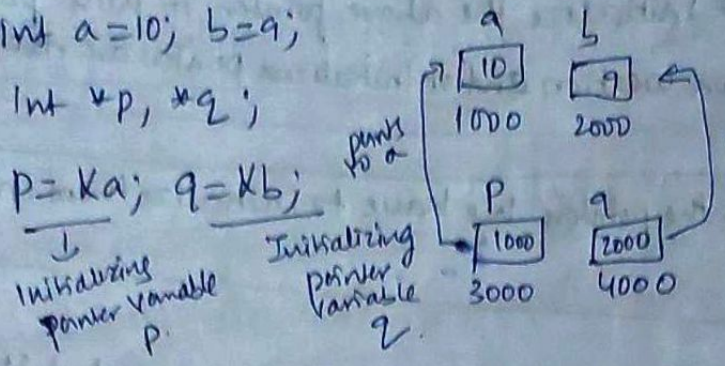
What should be the size of pointer?

Size of the pointer depends on the machine / it depends on the Compiler.

One 16 bit compiler / machine – 2 bytes.

32 bit compiler / machine – 4 bytes.

```
int a=10; b=9;
int *p, *q;
p=&a; q=&b;
```



→ here a will take 4 bytes, so in 4 bytes like address will be 1000, (the address of starting byte) is 1000) then 1001, 1002, 1003 thus will be for a variable.

but address will be base address which will be taken.

b will take 4 bytes and its base address is 2000

→ We have taken 2 pointer variables p, q, these variables also occupy some space in computer memory and obviously for these variables also there will be some memory assigned. and suppose the addresses of p variable (pointer) is 3000 and q pointer is 4000.

→ If we don't initialize them they will not point to anything.

→ We will initialize as p=&a; → now address of a is 1000, and it will be stored in p, so now 'p' points to 'a'

→ q=&b; → now address of b is 2000 and it will be stored in q, so now 'q' points to 'b'.

int *p, *q; → means whatever the address this p will contain, or p will store the address of a variable whose datatype is int here.

⇒ & → operator, meaning of this is to take the address of any variable.
↳ means it will return the address of variable what we specify. here &a or &b and 'a' address assigned to p 'b' " " to q.

generally we use this in scanf function.

We call this & operator as reference operator.

Accessing the values of a &b:

If i want to access the values of a and b?

How you can print?

There are multiple ways to print:

can we write like:

p = &a, &b; → Is it correct?

yes it is correct, but here, two operators are there one is comma and one is assignment operator, and precedence of assignment operator is higher than comma operator means, first assignment will be executed means p=&a, what ever is to the right side of '=', that will be assigned to this p, means 1000 will be assigned to p not address of b, this is the meaning of this statement.

→ if you write like p=(&a, &b); → In this ( ) are having higher precedence. so these will be executed before = (assignment operator).

→ Working of comma is, First operand will be evaluated and discards the result, we will evaluate the second operand and return the value. hex 2000 will be stored in p.

More than one pointer can point to a single variable.

q=&a; p=&a; In both pointers the address of a will be stored i.e. 1000 here. i.e both q & p are pointing to 'a'.

We can store like this also

int a=10, b=a;

p=&a; or p=&q; - first address of 'a' will be stored

q=&b; p=&b; then address of 'b' will be stored.

Suppose, i want to, printf("value of a=%d", a); then p points to b. now address of a is not pointing by any pointer variable.

↓

what it will print = 10.

We can access the value of above using pointers too:

because pointer p have address of "a variable"

so using this pointer also we can obtain the value of a.

Realtime example:

Suppose there is a person named ankit whose address is something like H.NO---- Soon and suppose pointer is some friend of Ankit named pulkit, if let us say i am third person & i dont know address of Abhi and i want to access Abhi and want to go to abhi's home, but i know pulkit, so i can go to pulkit and he is having the address of abhi and in pulkit we have like h.no---- address of abhi. so using pulkit i can reach abhi.

accessing the value of variable using pointer:

So using this pointer p, we can access the value of 'a'.

How to print the value of a using pointer p:

int a=10, b=9; int *p, *q; p=&a; q=&b;

```
printf(" value of a = %d", p);
```

→ If you print simply p, it will print 1000, in p we have the address of a i.e. 1000 and in q we have the address of b i.e. 2000

If you print like above, it will print the address of 'a' i.e. 1000 as p contains address of a.

So for printing the value we use dereferencing | Indirection operator.

```
printf("a = %d", *p);
```

If you give *p ⇒ it is going to print the values at this address.

Simply Indirection Operator is used to print the value at address.

*p ⇒ means value at p. i.e. value at 1000 = 10 will be printed.

We can write like:

```
*p = *(&a);
  ⇒ *(1000) = 10 will be printed
```

If you want to print address of 'a' using pointer:

```
printf("address of a = %d", p); → decimal.
printf("address of a in hexadecimal form = %x", p); hexadecimal form.
  "        "              "            "   ", &a);
```

If you want to print the address of p i.e pointer

```
printf("address of p = %x", &p);
```
prints 3000

Question:

```
int c;
c = *q;
printf("c = %d", c);
```

⇒ c = *(&b) = *(2000) = 9 will be printed.

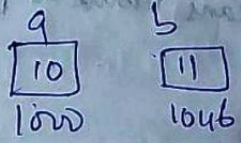
```
int c; int a=10, b=9; int *p, int *q; p=&a; q=&b; c=*q;
*p=20;
printf("a = %d", a);
printf("%x", p);
```
} ⇒ Now what it will print?

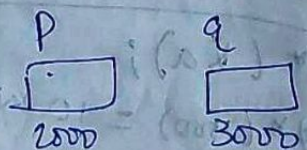
⇒ Pointer Assignment:

pointer Assignment means assigning the value of one pointer to another pointer.

We will see that with one example / simple program here.

```
int a=10, b=11;
int *p, *q;
```





Here pointer q we are not having anything as we have not initialized these pointers and these are pointing to unknown locations.

Note:
= * → is what ? Indirection Operator, means it will give the value at this address.
of *p ⇒ means it will give value at p i.e

Wrap-

But when you write `int *p;` i.e. `datatype * pointername;`

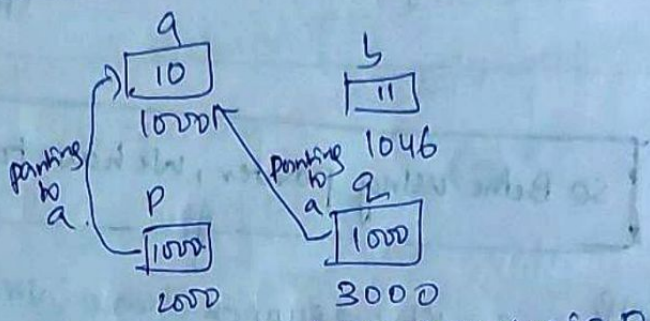
means, here * will tell the compiler that it is a pointer. and here it is not acting as indirection / dereferencing operator.

Wrap

→ Because here we are declaring pointer.

int *p; or int *q; means "*" will tell the compiler that it is a pointer and not acting as indirection operator, because we are declaring the pointer and is going to tell the compiler that "p and q are pointers and these are going to contain the address of variables of int data type".

⇒
```
int a=10, b=11;
int *p, *q;
p=&a;
q=p;
```
⇒ If is write like this what ever the value in p (assignment operator. (it is correct in C language not illegal), we have 1000 here and this will be assigned to q i.e. now q is also pointing to 'a' now.



If you print now value of a

```
printf("a=%d", a);
printf("a=%d; d=%d", a, *p, *q);
```

here it will act as indirection operator, so If you write like this's it is going to print 10 at both cases

If you write something like below:

```
int a=10, b=11;
int *p, *q;
p=&a;
```

`*q = *p;` // may be it will give error / warning

because,

*q means "value at this address", but i haven't initialized 'q' yet, because it is pointing to some unknown location.

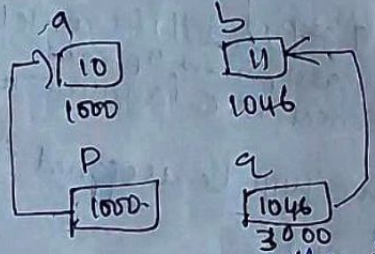
But we want to access, value at this address and in 'q' we dont have anything, so we don't know the value at this address. We are going to access "Illegal memory location" so it will give some error. (like your program name.exe has stopped working or some error).

So Before using pointer, we have to initialize the pointer.

Wrong ⇒ `q=&b;` ⇒ suppose i have initialized like this.

⇒ Now;

```
int a=10, b=11;
int *p, *q;
p=&a; q=&b;
```



`*q = *p;` ⇒ Now if you assign like this it is right, but what will happen?

Explanation:

*p ⇒ means *(&a) = *(1000) ⇒ value at 1000 = 10

i.e. now this 10 value we will access like this and now this '10' will be assigned where? at "*q"

means value at this address i.e *q = *(&b);

⇒ *q = *(1046) = 11 (we are accessing this value here.

so now this 10 will be assigned to b variable, so now in b we will have value '10', instead of 11 how because of this statement.

In this case we are accessing the values, in the previous case we were accessing the pointers, and these two are different things which we need to take care.

// Now if you print like below.

printf("value of a = %d %d %d", a, *p, *q);

a will print 10

and *p ⇒ value at &a ⇒ value at 1000 ⇒ it will print 10

*q ⇒ value at &b ⇒ value at 1046 ⇒ it will print 10

But if you write q=p; // here address of p is assigned to q

i.e address is stored here.

If so now if you print *q ⇒ it will print 10.

Extended concept of pointer i.e. pointer to pointer (Double pointer).

This concept of "double pointer" (or) "pointer to pointer", is very important when you are dealing with pointers.

Recap:

```
int a=10;
int *p; // declaring a pointer.
p=&a; // Initializing a pointer.
```

or we can write like: int *p=&a;

`int *p=&a;` ⇒ This means suppose in memory

→ it's long tape of bytes.

We can draw it horizontally / vertically, it is a long tape of bytes and (1byte, 1byte, ---)

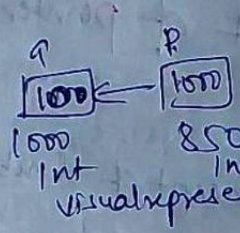
⇒ int a=10; → Somewhere memory will be allocated to this, i.e. 4 bytes, somewhere.

⇒ int *p=&a;

*p ⇒ means it is a pointer (it is declaration and initialization of the pointer.

So * will tell that 'p' is a pointer to the compiler, but for 'p' also some memory will be allocated, and size of pointer on 32 bit machine is 4 bytes. Suppose 'p' is allocated 4 bytes somewhere in memory and in this 'p' we store address of a is 1000 to 1003, Now in 'p' it will store the base address of a i.e. 1000

⇒ Now 'p' is what →

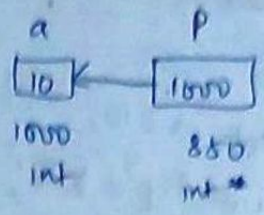


"A pointer to integer datatype"

means it can hold the address of a variable whose datatype should be integer (it cannot point to float ... or such)

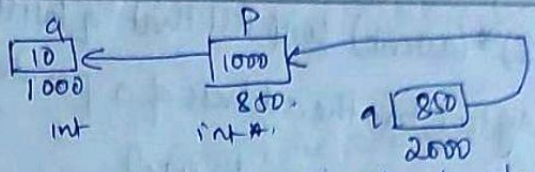
→ Now we need to understand what is pointer to pointer.

pointer to pointer / double pointer:



Def: pointer to pointer means or a double pointer is a special variable or a special pointer which is going to store the address of another pointer. (not any ordinary or normal variable).

> Double pointer is going to store address of another pointer variable.



⇒ int a = 10;

int *p = &a;

Suppose if, take another pointer 'q' and q is going to store (it is going to store another pointer's address, it is a pointer to pointer i.e.

Suppose it is pointing to p (another pointer variable).

→ It is not pointing to a normal variable, it is pointing to a special variable (pointer of another variable).

→ i.e. Why it is double pointer.

→ It is going to show level.

→ so in q address of p i.e. pointer will be stored.

→ and q is also a variable and it will also be allocated some memory location suppose here 2000.

So q is here it is, **pointer to a pointer to variable.**

q → p → a.

q points to p (pointer) → it points to a (normal variable).

↓ this holds address of pointer.

↓ this holds address of a variable

**

→ How to declare pointer to pointer / double pointer:

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

If I want to declare q and in q i want to store address of p(pointer)

Imp First thing is 'q' is also a pointer and '*' asterik will tell the compiler that (when you declare) it is a pointer, definitely one (1)*(asterik) will tell that q will be a pointer and q is going to store the address of a pointer variable. so which type of datatype we have to use?

Whose address this q is going to store.

so q is going to store the address of p, so data type we write is "int *".

i.e. Why we have 2 asterik's in double pointer.

`int **q;` → this is how we declare a double pointer.

Im `int *p;` → here asterik will tell compiler that it is a single level pointer and int will tell datatype of the variable address of the variable what p is going to hold.

`int **q;` →

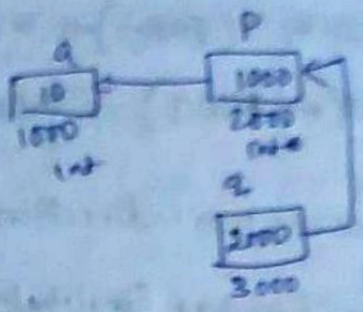
↳ this asterik will tell to compiler that it is a pointer i.e q is a pointer because it is storing address, but q is storing the address of a pointer variable so datatype we have to mention here i.e. we have to tell the type of address it is going to store

and that should be int **.

→ We can say it as a two-level pointer.

How to initialize a double pointer

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



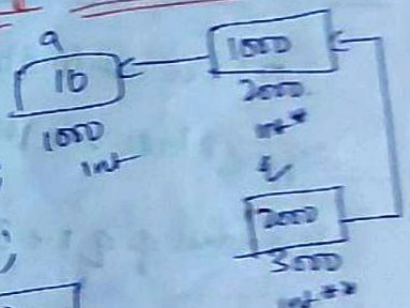
If you write here:

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

→ It will be illegal. Because a is not a pointer variable and q is a double pointer.

So according to the rule, q should contain the address of a pointer variable only.

Printing the value of a (normal variable) using double pointer



Various ways:

1. printf("value of a = %d", a);
2. printf("value of a = %d", *p);
3. printf("value of a = %d", **q);

↳ If you write here *q in printing,

*q ⇒ means *(&p) ⇒ *(2000) ⇒ this will tell value at this address i.e. *(2000) ⇒ value at 2000 is 1000 which is address of a.

But we want value of a, so we want value at 1000 (again address) so again we need to put asterik before this i.e.

**q;

$**q \Rightarrow *(*q)$

$\Rightarrow *(*(\&p))$

$\Rightarrow *(*(2000)) \Rightarrow *(\text{value at address } 2000) \Rightarrow *(1000)$

$\Rightarrow *(1000) \Rightarrow \text{value at } 1000 \Rightarrow 10$

So Here we use two indirection operators.

you can say it as the two indirection pointer, because obviously it is a two-level pointer.

so for accessing 'a' value first from 'q' we go to 'p' then 'a'.

one * we use we go from q → p and another * we use from p to a. So we can use double pointer even to access the value of 'a' (normal variable).

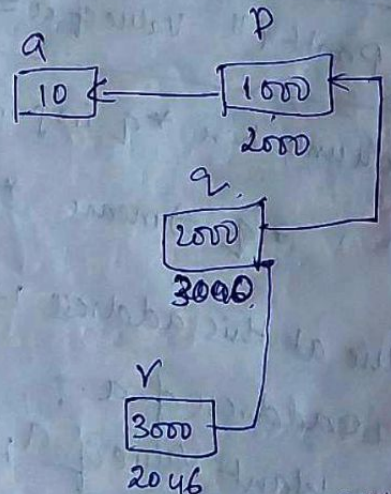
Moving

We can increase this level also:

If i want to store the address of q.

suppose i take one more variable 'r'. and here i want to store the address of q i.e 3000 here for example, and address of 'r' is 2046 and r points to addres of q.

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



Now * is what?

it is a pointer variable

and it is holding address of another pointer variable q and q is another pointer variable p and p hold address of normal variable a.

r => pointer to pointer to pointer to variable.

In r => `int ***r = &p;` if you want to store address of p in r this will also be illegal because

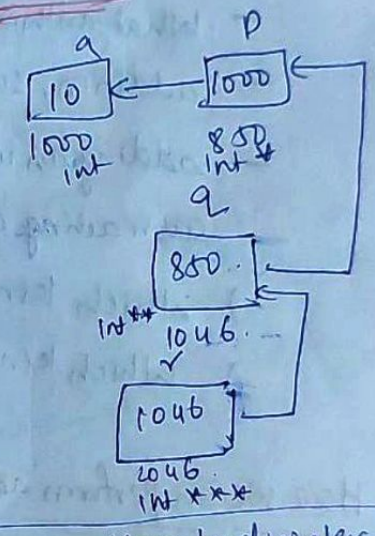
it is a three level pointer, it can store address of a two-level pointer according to the rule.

If it is a two level pointer, it should store address of a one-level pointer according to the rule.

If it is a one-level pointer, it should store address of 0 level i.e address of another variable.

Accessing a variable 'a' using 3-level pointer (r) here:

```
printf("value of a = %d", ***r);
```



***r => *(*(*r)) => *(*(*(1046)))
=> *(*(850))
=> *(1000)
=> 10

This is how we can access a variable value using three level pointer.

Changing the value of 'a' (normal variable) using double pointer and three level pointer.

**q = 25; => now after this if you print the value of 'a' i.e 25 will be printed.

*(*q) => *(*(*p)) => *(*(850)) => *(1000) => value at this address = 10

This value will be changed to 25 as we have initialized 25 now.

if you write like this:

`*q=25;` ⇒ gives error.

*(800) ⇒ it contains address i.e. 1000

now you want to store 25 a value here and directly we cannot assign any integer value here in a pointer this will be invalid.

`****r=25;` ⇒ We are changing the value of a using r.

⇒ **pointer arithmetic:**

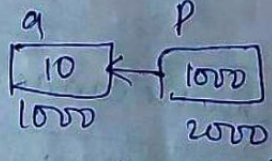
Contents:

- What arithmetic operations can be performed on pointers.
- addition, subtraction, increment, decrement
- adding an integer value to a pointer.
- subtracting an integer value from a pointer.

→ Which kind of operations are valid on pointers and

→ Which kind of operations are invalid on pointers.

How we can perform addition on pointer:



let us take one variable:

```
int a=10;
int *p=&a; // took a pointer variable and
```

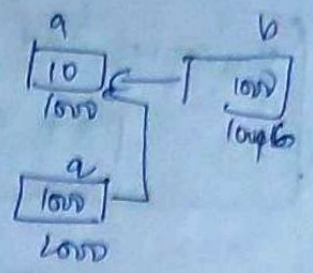
Stored the address of that variable 'a'.

→ a contains value 10 and pointer p contains the value which is address of a i.e. 1000 and 'p' also contains its own address i.e. 2000 (here) for example.

But actually the addresses will be represented in hexadecimal

We will take one more pointer variable.

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



V imp

⇒ If you want to perform like below:

p+q ;⇒)

If you want to add two pointer variables it is not possible in 'C', it is Invalid in 'C'.

You cannot perform addition on two pointer variables.

This is Invalid.

How you can perform addition on pointers then ⇒)

Here the case is:

You can add integer to a pointer.

you can add any integer value in pointer

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

if you want to add p+1 ⇒ this is fine.

p+5 ⇒ this is also fine.

p+0 ⇒ this is also fine.

Here we are adding integer value in a pointer variable.

This is correct,

You cannot add two pointer variables like
p++ or p+q where q & p are pointer variables.

⇒ Ex: int a=10;
int *p=&a;
p=p+2; // if you write like this suppose,

p is a pointer variable and pointer variable p is having 1000 (address of a), you will add +2, and we think that the answer will be 1002 (for our sake we took address as 1000 but it will be in Hexadecimal form. Or it would be any unsigned integer value).

Note: ⇒ But 1002 is wrong.

Now how this addition will be performed →

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

Suppose int is taking 4bytes in a 32bit Compiler.

If a is something like below

a

		10	

1000 1001 1002 1003

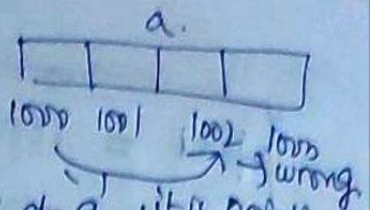
→ 10 will be converted into 0's & 1's of 32bits and stored.

p will contain the base address i.e. 1000 of a variable.

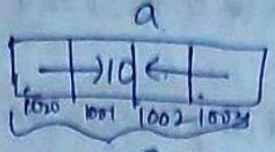
If we add +2 means now pointer will point to 1002 →

does it makes any sense like this? NO

It is illogical.



pointer is representing the complete address of a, it is pointing to complete variable.



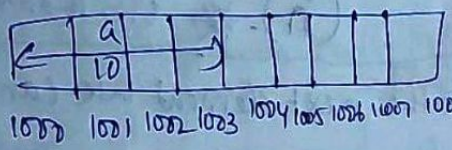
↑ p↑ points to complete address of a or complete a variable.

What is the logic of p+2 and pointer pointing to 1002? NO.

It is illogical

Now what does this p=p+2; ❀ mean?

In pointer, a is holding 4bytes a memory location and after that also we have memory like below.



-- -- So on.

Let us add first p=p+1; here.

wrong: it is not 1001

wrong: It will point to the next memory location i.e. 1004

Complete 4 bytes i.e from 1000 to 1003 will be for a and if you add +1 then it will point to the next memory location i.e. 1004.

Ultimately how many bytes will be added?

4 bytes will be added.

you are adding 1 to p, but 4 bytes has been added.

So what should be the formula?

p+1 = 1000 + 1 * size of datatype.

= 1000 + 1 * 4 = 1004.

This is how we add any integer value in a pointer.

Now in p we will have 1004.

p

Now suppose i want to add here 2 to p!

p+2 = 1000 + 2 * size of datatype.

p+i = address of the variable what the pointer is holding + integer value * sized datatype.
(p) + (n). * size of datatype.

It will point to the next element we can say.

⇒ There is no need of adding integer value and all here because we are taking only a single variable.

→ and here we don't have any other element or may be we have something we don't know what is there in that memory location.

→ if you print after printing p = p+2; like below.

printf("%d", *p);

here now pointer is pointing to 1008 location.

*p ⇒ *(1008) ⇒ value at 1008 what value we have → we don't know.

So if you print the value at p now i.e. value at 1000 → it will be a garbage value and it will be printing garbage value.

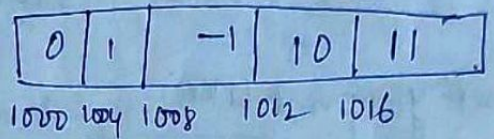
→ So there is no need to add this integer value to pointer when we are dealing with variables.

→ This will be useful when we deal with arrays.

Let us discuss one complex example in which we take array.

```
int a[5] = {0, 1, -1, 10, 11};
```

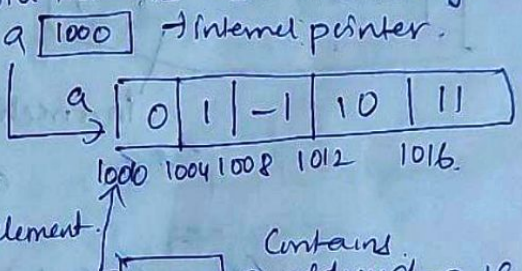
we have declared an array of size 5 and initialized the values.



and total how many bytes it will take is $5 \times 4 = 20$ bytes of memory.

```
int *p = &a[0];
```

a is an internal pointer which holds the address of first element.



and here now p is holding &a[0] i.e. 1000 this also points to first element of array.

But a is constant pointer. ⇒ we cannot do something like a++ or a--;

Imp.

But with pointer p we can do this p+1, p+2, but with a we cannot do addition like a+1, a+2--- (illegal thing)

a is constant pointer → It will always contains the Base address of array. It cannot be moved.

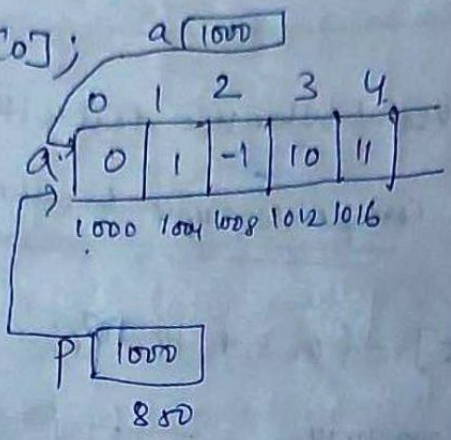
We cannot change the value of a whatever it contains.

The above is the visual representation.

But with P (pointer) i.e another pointer you can move. and can change the value (address).

Ex: int a[5] = {0, 1, -1, 10, 11};

int *p = &a[0];



Now if you print like:

printf("%d", *p); ⇒ What o/p you get.

Indirection Operator means we get value at an address.

⇒ In p we have 1000 which is address of a and value at 1000 = 0, so it will print 0 the first element.

Ex: p = p + 2 if we do?

int a[5] = {0, 1, -1, 10, 11};

int *p = &a[0];

printf("%d", *p);

p = p + 2;

printf("%d", *p); ⇒ p is pointing to 1008 and *p ⇒ *(1008) ⇒ Value at address 1008 = −1 will be printed.

When we do p=p+2; then it is pointing to the element "-1" (3rd element).

First it was pointing to 1000 (first element) then if you do +1 it will point to 1004 (second element), then if you do +2 it will point to 1008 i.e to the element -1.

two positions it will move further.

p=p+2; ultimately you can find the address also and in that address we will get value. (and now p points to 1008) now (it will print -1).

⇒ Now ⇒ p= &a[2]; i.e Now p contains address of a[2]

If p=&a[2]; in starting p=&a[0]; and in p if we add any integer no 'n'.

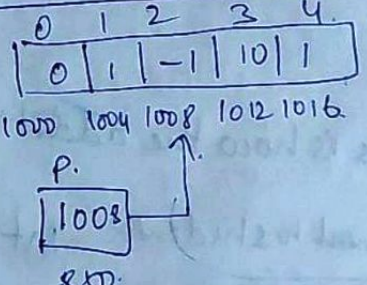
p+n; ⇒ In 'p' if you add any integer no 1, 2, 3, 4 ---

we can write like:

p+n = &a[0+n] } this is in general we can write.

p+2 = &a[2];

We can also access the array elements using pointer.



If you want to print (-1).

In this array we can print like.

printf("%d", a[2]);

(or).

Using pointer variable. after doing p=p+2;

⇒ printf("%d", *p);

Now do p=p+1; and if we print like.

printf("%d", *p); ⇒ What will be the answer-?

Sth a misunderstanding:

Now p is pointing to 1008 and p+1 means it points 1012 ⇒ NO

We have written p+1 only but we are not storing this value in p or anything. i.e we are adding +1 to p but we are not storing the updated value in p.

⇒ If yo do just p+1, it will not reflect p and still p will be pointing to 1008 location only.

But if you do p++; ⇒ this is fine. that is increment Operator ultimately it will add p+1.

⇒ If you have to write explicitly then you have to write

p=p+1;

⇒ Now p points to 1012 and *p means

*p = *(&a[3]) = *(1012) ⇒ value at 1012 = 10 will be printed.

This is how the addition will be done.

Assignment to students: int a[5] = {0, 1, -1, 10, 11};

int *p = &a[0];

printf("%d", *p);

p=p+2;

printf("%d", *p); p=p+1;

*p = 2; printf("%d", *p); → Op-?

In array it makes sense when we move pointer because we have more elements.

While taking one variable, in this case it doesn't make any sense to increase the printer like p+2 & p+3. It is illogical.

Generally we perform arithmetic operations on pointers when we use / deal with arrays.

Note:
In the above if you do p= p+10; here it will print garbage value because size of array is only 5 and elements are only 5 here.

Program:

a)
```
#include <stdio.h>
void main()
{ int a[5]={1,4,3,-8,0};
  int *p= &a[0];
} printf("%d", *p);
```

b)
```
#include <stdio.h>
void main()
{ int a[5] ={1,4,3,-8,0};
  int *p = &a[0];
  printf("value is: %d\n", *p);
  printf("address of element=%u\n", p);
  p= p+2;
  printf("value is :%d\n", *p); // element at 3rd position and
  printf("address of element is: %u", element at second index.
                                  p);
}
```

c) #include<stdio.h>

```
void main()
{
  int a[5]={1,4,2,-8,0};
  int *p=&a[0];
  int *q=&a[0];
  p=p+q; //error: invalid operands to
                binary + (have 'int *' and
  // we cannot add two pointer variables 'int *').
  printf("value is: %d\n",*p);
  printf("address of element is: %u\n",p);
  p=p+2;
  printf("value is %d\n",*p);
  printf("address of element is %d\n",p);
}
```

Op: gives error as we cannot add two pointer variables.

If you write like *p=34;

and if you print like printf("%d",*p);

It will print 34.

Ie- if p is pointing to 1008 and if let us say the value is 2 in that that value changes to 34 if you write like *p=34;

This is how additions performed.

⇒ Pointer Arithmetic (subtraction).

- How to perform subtraction on pointer variables?
- can we subtract two pointer variables?

Ans: yes we can subtract two pointer variables.

We cannot add two pointer variables, we can only add an integer to a pointer.

→ We can subtract an integer value from a pointer variable too.

Ex: `int a[5] = {0,1,-1,10,11};`

`int *p = a;` // simply we can store the name of the array also while initializing pointers using arrays because arrayname always points to base address of array;

// `int *p = &a[0];`

`int *q = &a[3];`

We took 2 pointers where one pointer (p) is pointing to `&a[0]` and pointer (q) is pointing to `&a[3]`.

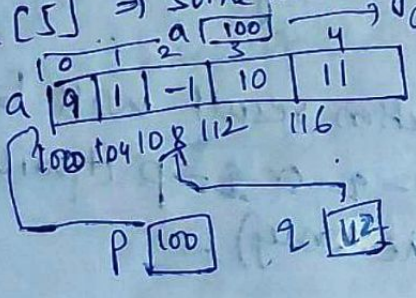
Now if we do `p-q;` ⇒ is it possible? yes it is possible.

`q-p;` ⇒ is it possible? yes it is possible.

`p-2;` ⇒ is it possible? yes it is possible.

so both the above cases are possible in subtraction.

→ If you declare `a[5]` ⇒ some memory will be allocated to this array.



// p have addrss of a[0] i.e. 100 and in q we have address of a[3] i.e 112, and a is constant pointer and internal pointer, so obviously we will have the base address of the array i.e 100. and by default int will take 4 bytes of memory, so the above are the addresses of all the elements of the array.

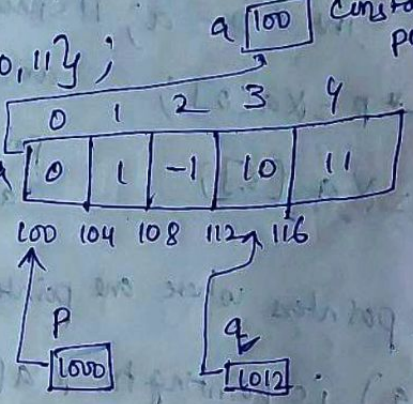
// Now if you want to findout the difference of p and q:)

How to find the difference of p and q?

// you have to declare a variable for storing the difference and difference value will be an integer value we get.

Program:

```
int d;
int a[] = {0, 1, -1, 10, 11};
int *p = &a[0];
int *q = &a[3];
d = q - p;
```



Note:

1. If you do p+2 ⇒ we get the result in the form of pointer. i.e if p=1000 and if you do +2 means you will move this pointer forward by 2 elements. i.e. it will forward or point to address 1008 means it is giving pointer only (or we can say the address).

2. But the subtraction result will be an integer value. (do p-q or q-p it will give integer value only).

VIMP

(3) But if you subtract any integer value like `p=p-2;` from a pointer, the result will be in pointer form i.e. address we will get or we can say the address of the elements two positions behind to the p (which it is pointing to).

// if you do:

d = q - p; // in the above.

in q we have 1012 and in p we have 1000

misunderstand? ⇒ 1012 − 1000 = 12 (wrong - we get)

VIMP ⇒ 12 doesn't make any sense.

VIMP ⇒ you have to divide this value by size of datatype.

i.e 12/4 = (3) → this means q-p will give you 3 and '3' means the elements between index 1 and index 3 i.e between p and q (1, 2, 3 not 0th index) 3 elements apart

So if you subtract q-p ⇒ i.e when you subtract two pointers, how many elements are there between those 2 pointers that we will get..

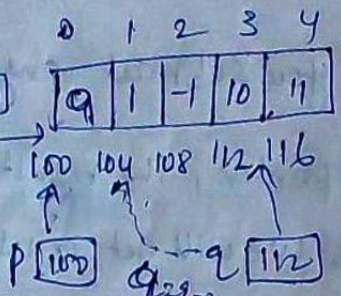
⇒ if you do ⇒ d = p - q; ⇒ 1000 − 1012 ⇒ we get −12 and divide by size of the datatype i.e 4 ⇒ −12/4 = −3. we get (−3) here, (back of q 3 elements).

⇒ But if you do q = q - 2; here we get a pointer (address).

= 1012 − 2 = (1010) ✗

⇒ means here it moves backward by 2 elements.

```
d=q-p;
q=q-2;
d=p-q;
p=p-1;
```



if you do q = q-2
q moves to 104 and
q points to 104

formula will be:

$$q - q = q - 2 * \text{sizeof datatype}$$

↳ address where it points to.

$$= 112 - 2 * 4 = 112 - 8 = 104$$

$$q - n = q - n * \text{sizeof datatype}$$

⇒ result will be a pointer.

if you do p=p-1 after d=p-q, it will be moved to a location behind 100 i.e. it will be 96 but if you access this location you don't know what is there in this location a garbage value will be printed.

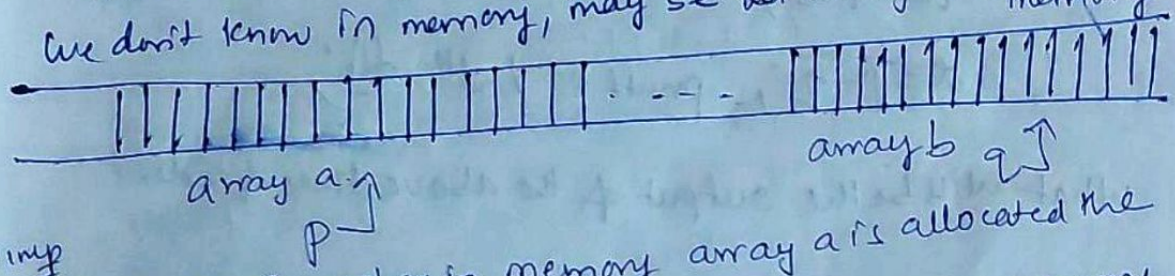
Better to move pointer within the range of array

If you move pointer beyond range, it will give some undefined behaviour.

⇒ If you are having two arrays let us say:

```
int a[] = {1,2,3};
int b[] = {0,1,-1,10,11};
int *p = &a[0];
int *q = &b[0];
```

if you do now p-q or q-p, it doesn't make any sense, it will show some undefined behaviour, because p is pointing to array a and q is pointing to array b. What is the use of finding the difference between these 2?! We don't know in memory, may be let us say following is the memory.



VVIMP

⇒ Let us say somewhere in memory array a is allocated the memory locations and array b is allocated the memory locations somewhere else.

⇒ p is pointing to array a and q is pointing to array b.

⇒ and in between array 'a' and array 'b', we don't know how many memory locations are there (or) what data is stored in those memory locations.

→ So what is the logic to find the difference between p & q when they are pointing to different arrays?

→ If both are pointing to same array then it is meaningfull to find the difference of these 2 pointers.

Program / Assignment to Shudents!

```
int d;
int [] = {9, 8, 3, -11, 10};
int *p = &a[0];
int *q = &a[4];
d = p - q;
printf("%d", d);
*q = 25;
d = q - p; printf("%d", d);
*p = 27;
q = q - 3;
d = p - q; printf("%d", d);
```

What will be the output of the above lines of code?

Basic pointer Example program:

```
void main()
{
    int a=10; int *p = &a; int **q = &p;
    int ***r = &q; *p = 12; **q = 17;
    ***r = 78;
    printf("a = %d\n", a);
    printf("a = %d\n", *p);
    printf("a = %d\n", *(*q));
    printf("a = %d\n", *(*(*r)));
    printf("address of q = %x %x\n", r, &q);
    printf("address of p = %x %x\n", &p, q);
}
```

programs on pointer subtraction:

a)
```
#include <stdio.h>
void main()
{ int a[5] = {2,4,3,10,-7};
  int *p = a;   // int *p = &a[0];
  int *q = &a[3]; // q is pointing to 3 index i.e. 10.
  printf("q-p = %d\n", q-p); // how many elements between q & p
  printf("p-q = %d", p-q); // 3 elements apart.
}
```

o/p: q-p = 3.
p-q = -3.

b)
```
void main()
{
  int a[5] = {2,4,3,10,-7};
  int *p = a;
  int *q = &a[3];
  printf("q-p = %d\n", q-p);
  printf("p-q = %d", p-q); printf("%d", *q);
  q=q-2; printf("value = %d", *q); 4
  p=p+2;
  printf("q-p = %d", q-p); // q is pointing to 4.
                           // p. points to 3.
  q=q-2;                   // -1 it will give.
  printf("Value = %d", *q);
}
```

Pointer Arithmetic (Increment / decrement)

Contents:

- How to perform increment and decrement on pointers
- post increment, pre increment, predecrement, post decrement.

Increment operations on pointers:

→ int a[] = {3, 2, 67, 0, 56}; // took an array
int * p; // took a pointer.
p = a; // this is valid, you need to write address of a[0].
// a is a constant pointer / array name contains
// the base address of array / base element addr.

But if you write:

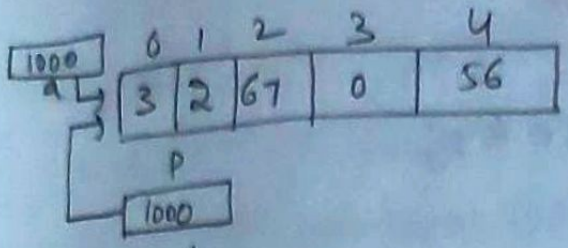
a = p; // you think that a is also pointer, as named the
// array acts as a pointer to this array.
// and p is also a pointer, you can assign this value.
// NO this is not possible, 'a' is what? it is a constant
// pointer to the array.
// name of the array is constant pointer to the array a[].

⇒ In 'a' you cannot assign any thing or you can't increase or decrease. i.e you can't do a++ or a--, you cannot do anything with arrayname.

→ Using pointer you can do this..

→ p = a; → is valid but a = p; is not valid.

→ Let us see the visual representation of the above array we have declared.



both a & p are pointing to the first element of this array. We can move this pointer p, but we cannot move this 'a', as it is constant pointer which always contain base address of array (1000)

How to increment ?

If you want to increment this p, suppose you write p++.

p++ or ++p means, We are moving one location forward.

p=p+1; but not only p+1; (here we are not updating the value).

Ultimately we are adding 1 to p, but 1 means we have to add, in p we have 1000, so we have to add ⇒ 1000 + 1 * size of datatype ⇒ 1000 + 1*4 ⇒ 1004.

So p++ ⇒ means it points to the next element.

again p++ ⇒ " " " " " " " so on.

Now!

```
int a[] = {3,2,67,0,56};
    int *p;
    p=a;
    p++;
printf("%d", *p); // * acts as indirection operator,
```

so ie value at the p address. ⇒ *p ⇒ *(& a) ⇒ value at address of a. i.e you will get 1000 address i.e. 3 ⇒ NO.

and now as we have incremented p++ and p is pointing to 1004 and now the value at this address is 2.

So 2 will be printed.

Now a small change in the above code:

⇒ Post increment:

```
int a[] = {3,2, 67, 0, 56};
int *p;
p = a;
printf("%d", *p++);
```

here it will return 1000 for p++ and *(p++) is *(1000) = 3 we have combine

// If we write like above *p++ ⇒ this means // increment and asterik.

Now What output we get here-)

p++ ⇒ means it is post increment.

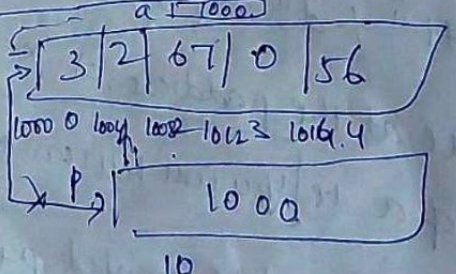
and at starting p is pointing to 1000 (address of a).

What is the functioning of post increment-)

First of all it will return the value and then it will do ++ (increment).

⇒ It means it will return the original value if post increment is there and then it will increment.

-) original value of p is 1000, so it will return first of all 1000 and now it will increment p by 1. i.e; it means not 1001 but it is 1004. (as p is pointer).



now p will be pointing to 1004:

So when you print the value:

printf("%d", *p++); in the above case it will return 1000 only as it is post increment.

and *(p++) ⇒ *(1000) ⇒ 3 will be printed.

and p will be pointing to address 1004.

and after the above printf statement if you print like below:

printf("%d", *p++); ⇒ Now it will print 2 but p points to 1008 location.

⇒ ++ VVV IMP. if you write like *(p++) in printf statement.

You will think that first it will increment and it will give value 2. is wrong.

→ first it will evaluate p++, yes it will evaluate p++ first, but what is the functioning of p++, this is post increment so first of all it will return the original value and then it will increment. So here the original value of p is 1000 and then p will print to 1004.

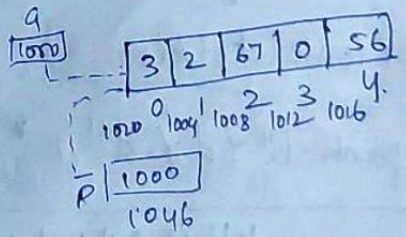
VVV IMP.
→ and the * (asterik will be performed on the previous value not the modified value i.e. on 1000 not on 1004.

⇒ so again *(1000) = 3, and it will print 3 only.

⇒ In the both cases i.e. *p++ or *(p++) it will give 3 only.

Pre increment:

```
int a[] = {3, 2, 67, 0, 56};
int *p;
p = a;
printf("%d", *++p);
```



If you write like: `*++p` ⇒ then what will happen?

Now ++p is what pre-increment.

What is the functioning of pre-increment?

The functioning of pre-increment is first of all it will increase the value, modify the value and then that modified value will be returned.

⇒ ++p ⇒ this means it will increment the value of p and in p starting we have 1000 and it was pointing to element 3 and now after incrementing it points to 1004 i.e. it points to element 2.

⇒ Now if you print like *p++ ⇒ it will be printing updated value i.e. *p++ ⇒ *(1004) ⇒ 2 will be printed.

Both if you print like *p++ or *(p++) ⇒ both will print 2 value only.

Note:

⇒ We can see in *++p or *p++, we have two operators one is "*" and another one is "++", and both these are having same precedence, but associativity is right to left, so first operator we have to perform is "++" and then "*".

If I write like below: ← R to L.

```
printf("%d %d", *p++, *p++);
```

What output you will get?

In the above case which will be evaluated first?

printf("%d %d", *p++, *p++);

Second this will be evaluated ↵ ↳ First this will be evaluated (LTR)

So this *p++ is post increment and initially p is holding 1000 so p++ will return original value i.e. 1000 and *(p++) ⇒ *(1000) i.e. Value at 1000 ⇒ 3 and now p will be incremented now it will point to 1004.

⇒ Next second *p++ will be evaluated.

i.e. Now original value of p is now 1004 so value at 1004 is i.e *(1004) is 2 so it will print 2 and now it will increment p++ i.e. it will point to 1008.

⇒ when you print like below:

printf("%d %d", *p++, *p++);

it will print 2 and 3.

Post decrement:

Ex: int a[] = {3, 2, 67, 0, 56};

int *p;

p = a;

printf("%d", *p--);

*p-- ⇒ it is post decrement,

Now what is post decrement?

P.T.O

What is the functioning of post decrement?

The functioning of post decrement is, first of all the original value will be returned and then decremented.

So `printf("%d", *p--);`

In *p--; ⇒ first of all original value of p i.e. from p all 1000 will be returned i.e. now *(p) i.e. value at p means value at 1000 i.e. 3 will be returned and after this p--; now p is pointing to (p should move one position backward means 996, now what value we have here?

We don't know, so after this if you print like:

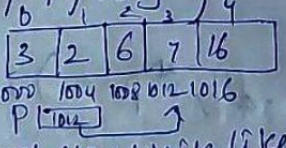
printf("%d", *p); → it will print garbage value.

because it is moving out of the range of the array.

Note:

We should take care of certain cases, we shall explain it by taking an example.

```
int a[] = {3, 2, 6, 7, 16};
int *p;
p = &a[3]; // If you write like this p will
```



point to 1012 i.e in p we have 1012 address

Now if you print: *p--;

original value of p now here is 1012, and this will be returned first and *(p) i.e. value at 1012 i.e 7 will be printed and now it will decrement by 1 i.e now p points to 1008 and after this if you print like

printf("%d", *p--); ⇒ it will print 6 and now after this p points to 1004.

pre decrement:

```
int a[] = {3, 2, 67, 0, 56};
int *p;
p = &a[3];
printf("%d", *--p); // this is pre-decrement
```

What is the functioning of pre-decrement?

The functioning of pre-decrement is, first of all the value will be decremented / modified then it will return the modified value.

→ now p is pointing to 1012 at original and after --p it will point to 1008 and now *--p; ⇒ that is it will print the value at *&--p; ⇒ *(1008) ⇒ 67 will be printed.

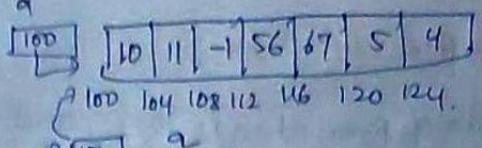
Question:

printf("%d %d %d", *--p, *--p, *--p);

--(*p); , (*p)++; (*p)--;

++(*p); -?

Program:




```
Void main()
{
    int a[] = { 10, 11, -1, 56, 67, 5, 4 };
    int *p, *q;
    p = a;
    printf("%d\n", *p);
printf("%d %d %d %d\n", (*p)++, *p++, *++p);
    q = p + 3;
    printf("%d", *--p + 5);
    printf("%d", *p + *q);
}
```

Note!

When you combine the increment, decrement operators, as well as this indirection operator, these type of questions will be very tricky in interviews.

→ If we know the working of this increment and decrement and *, operators and their associativity and precedence, (indirection) then it will be easy to understand, they are not much tough.

→ We will discuss these tricky questions with the above program: what output you will get and how you get that output.

Explanation of the program:

→ We have declared and initialized an array of 7 elements.

→ we have declared two pointers p, q and p is pointing to a and base address of a = 100.

→ if we print *p ⇒ *(&a) ⇒ *(100) ⇒ 10 will be printed.
(here * is indirection operator / dereferencing operator which means it will give value at that address).

→ Next line we are printing 3 things:

(*p)++, *p++, *++p ⇒ Right to left associativity.

Here we are having operators like increment (++), *(indirection operator and ( ) and these operators have right to left associativity.

*++p will be executed first (right side).

and this is pre increment, so p value will be modified and now it will be incremented by 1 means 100 + 4 = 104 now it points to 104 and *(++p) ⇒ *(104) ⇒ 11 will be printed.

Next *p++ ⇒ this is post increment, so first it will return the original value whatever is there in p and in p we have 104 so *(104) = 11 will be printed and now p will be incremented by 1 i.e. it will point to 108.

Next (*p)++ ⇒ here *p is in brackets and we have ++ brackets are having high precedence, so (*p) will be executed first not ++.

so *p ⇒ means *(p) ⇒ *(108) ⇒ it will print -1

WWINP.

and because it is post increment, it will return the value whatever is there in p and it will increment the (-1)

Value not the address ⇒ this we need to take care.

*p = -1 now and (*p)++ ⇒ and this "++" will be applied to (-1) value not on address.

so it will become -1+1 = 0 so value at 108 will become '0' (instead of -1).

Now if you print like *a[2] ⇒ it will print 0.

or *p it will print 0 ⇒ *(108) = 0.

⇒ Next is q=p+3;

Now q is a pointer and p+3 means, in pointer we are adding 3 it means, we will traverse forward 3 positions. so p will be pointing to 120 and in "q" whatever the address now p will be pointing to will be stored in q, so q will be pointing now to 120 memory location of a.

⇒ Now we are printing *q-3;

Obviously * have high precedence than subtraction operation, so first *q will be executed i.e. whatever the value we have at q i.e *q ⇒ *(120) ⇒ 5 and then -3 ⇒ 5-3 = 2 will be printed.

⇒ Next *--p+5 ⇒ rather than +, * and -- have higher precedence, but both * and -- have same precedence, and associativity will be from right to left, so first --p will be performed. i.e. predecrement will be executed first, so p is pointing to 108 and now after decrementing it becomes 104 and *--p ⇒ *(104) = 11 and now 11+5 = 16 will be printed.

⇒ now if we print *p + *q ⇒ obviously * ax having high precedence over "+".

*q ⇒ and q is pointing to 120 i.e. *(120) ⇒ 5

*q ⇒ p is pointing to 104 i.e *(104) = 11

and *p + *q = 11 + 5 = 16 will be printed.

Assignment to students:

```
void main()
{ int a[] = {10, 11, -1, 56, 67, 5, 43};
  int *p, *q;
  p = a;
  q = *a[0] + 3;
printf("%d %d %d", (*p)++, (*p)++, *(++p));
 printf("%d", *p);
 printf("%d", (*p)++);
 printf("%d", (*p)++);
  q--;
printf("%d", (*(q+2))--);
printf("%d", *(p+2) - 2);
printf("%d", *(p++ - 2) - 1);
}
```

pointers and Arrays:

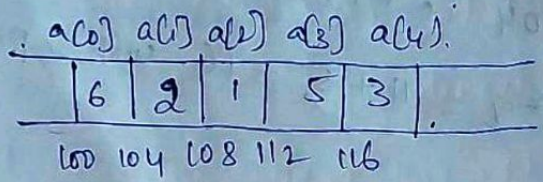
⇒ Array name itself is a pointer, it will act as a pointer

⇒ How arrays can be represented in memory, How arrays can be declared and initialized are already discussed.

```
int a[5] = {6,2,1,5,3}; // Initialization of array at compile time.
```

↳ If you don't mention here size also fine, but we have to initialize the values then memory manager will allocate the memory for these elements.

How the above array is represented in memory



(already discussed)

→ If the memory manager has assigned the Baseaddress to array as 100, the next element address Wouldbe 104 because datatype is int and its size is 4 byte.

⇒ Index of array starts with '0'. (0-4) here.

Before understanding the relationship of arrays and pointers, you should know how pointers can be declared.

datatype * nameofthepointer

int *p; ⇒ p is a pointer and it is a pointer to integer.

We cannot say that the datatype of this pointer is integer.

pointer is a variable that stores the address of another variable

- so pointer always store address of other variable, and address is always in the form of Hexadecimal (Long Hexa decimal No).

→ so we can say *p (pointer p) contains an address of a variable and the datatype of that variable is integer.

So it is a pointer to integer not an integer pointer.

→ Similarly for float, char ... so on.

int *p; float *q; char *c; ⇒ all these pointers will store address of a variable and the datatypes of those variables will be correspondingly int, float & char respectively.

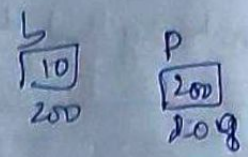
pointer is always a long Hexadecimal Number.

If you want to print the pointer, or the value of a pointer you can say or the value stored within that pointer that address in hexadecimal form then you have to use "%p" format specifier.

Ex: int b=10;

memory is a long tape of bytes and every byte have its own address, and somewhere in the memory b is allocated memory and suppose 'b' is allocated 200 as address, and at 200 value of b =10 will be stored and for pointer also somewhere the memory will be allocated & uploaded in memory.

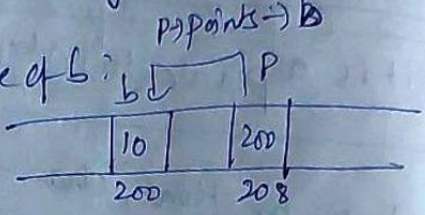
```
int b=10;
int *p;     p=&b;
```



Suppose b value is 10 and address is 200
and p is pointer and p's address is 204, value of p is always an address of other variable. like p=&b;

But you cannot write something like this `p=b;` → in this case value of b is 10, 10 will be stored in p. and pointer cannot store value, it can store only address.

Now if you want to print the value of b:




```
printf("%d", b);
printf("%d", *p); // using pointer.
```

↳ indirection operator / dereferencing operator.

*p ⇒ means value at this address (address of b).

*p ⇒ *(200) ⇒ value at 200 = 10.

To print value using pointers we use this dereferencing operator.

& — address of operator.

⇒ If you want to print the address of p, simply you can write.

```
printf("%d", p);
printf("%p", p);   // In hexadecimal form.
printf("%x", p);
printf("%u", p);
```

If you want to print you can use this.

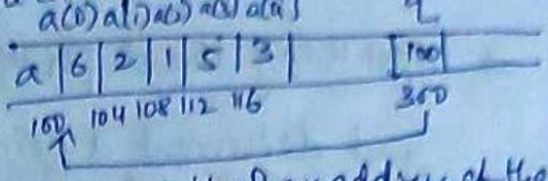
```
printf("%p", &b);
```

The above are the basics of pointers (recap).

How arrays are related to pointers.

→ arrayname itself acts as pointer.

int a[] = {6, 2, 1, 5, 3}; // sample array



→ So if you print 'a'

printf("%d", a); ⇒ It will print the Base address of the array. (100) will be printed.

printf("%p", a); ⇒ It will print the Base address of the array in hexadecimal form.

We will take an array:

int a[5] = {5, 2, 1, 6, 3};

int *q; // i took a pointer q and somewhere in memory q is assigned some memory. Let us say 300 is the address of q.

How to initialize this pointer?

q = a; → this is correct, but in the previous case

p = &b; is wrong.

↳ This is valid, because 'a' is arrayname and arrayname always contain the base address of array.

So this 100 will be assigned to q and this is fine, because q is a pointer and that will contain address of another variable, so in q we have Base address of a i.e &a[0] = 100.

⇒ If we write q = &a[2];

Now q will contain 108 and q pointer points to a[2] i.e to the element 1

If you print: printf("%p", a); ⇒ it prints 100
printf("%p", q); it will also print 100

⇒ Next thing about pointer is you can increment a pointer and decrement a pointer.

pointer contains an address, you can add some integer value in address and you can subtract some integer from an address that is allowed.

⇒ But address plus address is not allowed.
address * address is not allowed.

```
int a[] = {6,2,1,5,3};
int *q;
q = a;
printf("%p", a);
q++;
```

q = 100, q++ ⇒ 104; (q points now to 104. i.e if you now print *q ⇒ it will print *(104) = 2

q++ is what → it is containing address of a which is 100 (it is an integer variable (a)

So i.e. why if you do q++ then it will increase by how many bytes (depends on size of the datatype) it will increase by 4 bytes.

⇒ If it is a character pointer, in this case we can say if the value of q is 100, if it is a character pointer, it will increment 1 increase by 101. as character will take 1 byte.
in float → it increments by other byte. (size)

a++; ⇒ is not allowed you cannot change the base address of array.

a++ is not a valid operation.

q++ is a valid operation.

you can increment / decrement a pointer.

⇒ If you write q = a[2]; ⇒ is it valid ⇒ NO

because a[2] means a value is accessed i.e. 1 and it is being assigned to q, but this is not allowed bcz q is a pointer and it can store only address.

⇒ if you print ("%p", &a); it will also print base address of array.

But if you print ("%p", &q); it will print the address of this pointer.

Wing

In case of arrays we can say name of array = address of a

a will also print 100 and &a will also print 100.

But in case of pointers q ≠ &q. ⇒ q will print the value in this q and address of q will print the address of this pointer q.

We need to take care of the above things.

But address of a and a is same, because, a is an array.

In case of arrays it is allowed.

⇒ If you want to access any element of array then how you will access -)

printf("%d", a[2]); // If you want to print the value at a[2] it will print 7.

↳ this is the way we access.

⇒ we know the array name itself is a pointer and it contains the base address of the array. and if you want to access the values using pointers then we use dereferencing operator. (*)

So we can access the element of an array like below two.

a ⇒ contains base address i.e. 100

and a+2 if you do ⇒ In base address you are adding 2 it doesn't mean that you are adding the value 2 ..

In address if you add any integer value, according to the size of the data type u are using it will increment/ decrement.

→ In a+2 ⇒ +1 ⇒ means 4 bytes will be incremented
+2 ⇒ " 8 " " " "

printf ("%d", *(a+2)); // at 108 value is 1 it will print 1

rather than using a[2] you can use a+2 also. give 1.

⇒ Another method to access the array elements value is using pointers:

Suppose you want to access a[2] element using pointers.

Now q++ is done and q is pointing to 104 now if you do *(q+1) ⇒ Now q is containing 104 and q+1 is 108 and *(108) ⇒ a[2] = 1 will be printed.

If you have not done q++ and q is still pointing to 100 base address then you will do *(q+2) ⇒ it will print a[2] value i.e. 1

if you want to print the value at a[3]:

you can simply print like: a[3];

(or) you can write *(a+3) or you can write *(3+a)

In other words we can say,

$$a[i] = *(a+i);$$

a[1] = *(3+i); // *(i+3) or i[3] ⇒ you can write like this too.

*(a+i) we can write and even we can write this as *(i+a)

or i[a] also we can write.

*(i+a) = i[a] or

you can write even 2[a] ⇒ to print a[2] value i.e. 1.

⇒ If you want to print the value at 0th index.

printf("%d", a[0]); (or) it will print 6.

printf("%d", *a); // it will print 6, a will give base address.

printf("%d", *3); // *a ⇒ *(100) ⇒ value at 100 = 6.

If you write:

printf("%p", a+1); // a means it contains base address i.e. 100 if you add 1 in 100 it will point to the next element address i.e 104

** printf("%p", &a+1); // &a+1 means it will not give 104

&a means 100 and if you want +1 in &a then it will increment it by the complete array address. (5 elements size will be 5x4 = 20 bytes. So it will return 120 in this case.

Execute the above & check.

if you print: &a[0]+1 ⇒ this will give 104 bcz specifically assigning a[0] ⇒ &a[0] = 100 &100+1 ⇒ 104

so prints 104.

Now in printf ("%d", *a); then it will print value at 100 i.e. 6 it will print.

*(a+1) = ?

*a+1 = ?

Program: Initializing the array at runtime and printing the values using pointers.

```
void main()
{
    int a[5], i;      // declared an array of size 5
    int *q=a;         // one pointer q is taken and this
                      // q is containing &a ⇒ returns base address
                      // of array(100)
    for(i=0; i<5; i++)   // scanning the elements
                         // into array.
    { scanf ("%d", &a[i]);
    } scanf("%d", a+i);   (or)  rather than writing &a[i] simply we can write a+i.
```

// a will return address i.e. 100

in first case i value = 0, so a+i ⇒ 100+0 = 100 only

In second case i=1, so a+i ⇒ 100+1 = 104 (points to next element)

and so on.

Suppose you want to print the values of the array using pointers and user has entered values like 1, 2, 3, 4, 5 in a[0], a[1], a[2], a[3], a[4] respectively, we need to write another for loop

// printing the values.

```
for(i=0; i<5; i++)
{ printf("%d", a[i]);
        (or)
  printf("%d", *(q+i));
        (or)
  printf("%d", *(a+i));
```

q means contains BA = 100 and q+1 ⇒ 100+0 and *(100) = 1 ... (that prints)

(or) printf("%d", i[a]);
(or) printf("%d", i[2]);
}

Students: In scanf can we write scanf("%d", &a[i]);?

Array of pointers:

Just like we can declare an array of int, float or character, we can also declare an array of pointers.

The following is the syntax.

Syntax: datatype *array_name[size];

Example:

```
int *arop[5];
```

→ Here arop is an array of 5 integer pointers. It means that this array can hold the address of 5 integer variables.

→ In other words, you can assign 5 pointer variables of type "pointer to int" to the elements of this array.

program:

```
#include<stdio.h>
#define size 10
int main()
{ int *arop[size];
  int a=10, b=20, c=50, i;
  arop[0] = &a;
  arop[1] = &b;
  arop[2] = &c;
```

```
for(i=0; i<3; i++)
    {
    printf("Address = %d & value = %d\n", arrop[i],
                                          *arrop[i]);
    }
    return 0;
}
```

Expected o/p:

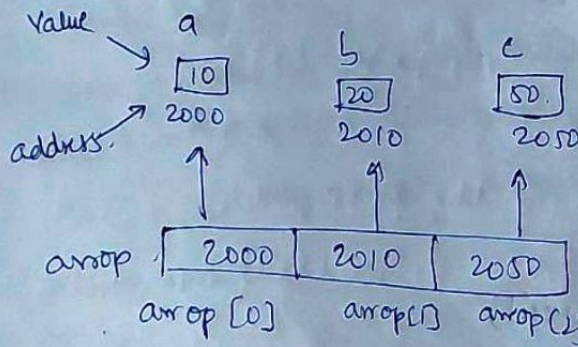
Address = 387130656 Value = 10

Address = 387130660 Value = 20

Address = 387130664 Value = 50.

How it Works:

- Notice how we are assigning the addresses of a, b and c.
- In the line 7 above in the program, we are assigning the address of variable a to the 0th element of the array.
- Similarly, the address of b and c is assigned to 1st and 2nd element respectively.



arrop[i] gives the address of ith element of the array.
So arrop[0] returns address of variable a,
arrop[1] returns address of b and so on.
To get value at address use indirection operator (*).

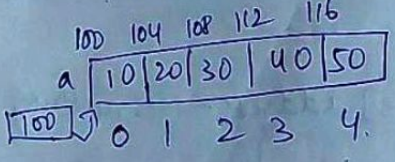
*arrop[i]

so, *arrop[0] gives value at address arrop[0],
similarly *arrop[1] gives the value at address arrop[1] and so on.

Array of pointers Explanation in a better way:

→ Generally we can declare a primitive type arrays.
→ It will be able to store character, integer, float values or any other kind type.
→ Array of pointers means an array can store addresses as elements.
→ This is what is an array of pointers and these pointers can be of any type. (It can be an integer pointer, a character pointer or float pointer --- soon).
→ Suppose you take one integer pointer.
First we will declare an array.

```
int a[5] = { 10, 20, 30, 40, 50 };   // we are declaring and initializing the array.
```



⇒ here array is assigned a memory and the values are stored in memory.
a is an internal pointer which holds the base address of array i.e. 100.

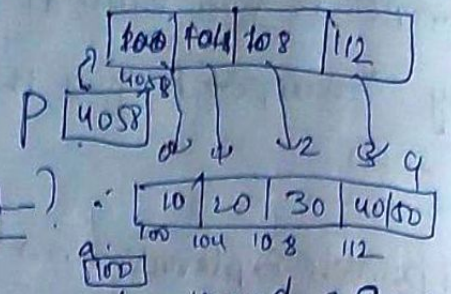
We are declaring an array (one more)

int *p[5] ⇒ this is not an integer type, it it is an integer type it holds integer data.

→ It is an integer *pointer type

so for p → memory will be allocated and p is a pointer pointing to some other location like 4058 and is pointing.

→ Now it can store addresses.



→ It can store addresses of what ?

Imp

→ It can be a different memory locations of a same array

→ or you can take 5 different arrays and every array address (base address) also we can store.

→ we can see in pointer array we are storing addresses of elements of a.

How to store ?

We use a for loop.

```
void main()
{
  int a[5] = {10,20,30,40,50};
  int *p[5];
  for(i=0;i<5;i++)   // we are declaring a variable i
  {                  //  i=0 & i<5 (index)
  => p[i] = &a[i];   // simply how can we store
  }                  // means a[0] address,
```

// a[1] address, a[2] address, a[3] address — it's nothing // but a[i] address i.e. &a[i] (i value varies from 0 to 4)

So address 100 will be stored in p[0], 104 in p[1], 108 in p[2], 112 in p[3]. Like this

so &a[i] is stored in p[i].

Using index we stored above.

Suppose we don't want to use index and want to use pointer arithmetic concept. (means we will not use index)

using pointer arithmetic if we store addresses of an array in a pointer array / Integer pointer as below.

```
for(i=0; i<5; i++)
  {
  *(p+i) = a+i;
  }
```

i=0;
a+i means ⇒ a holds base address i.e 100 and a+0 means 100 only and this will be stored in

*(p+i) i.e *(4058+0) ⇒ *(4058] i.e value at 4058.

i.e in p[0] it will store a[0] address.

i++ i.e: 1
a+i ⇒ a+i means here 100+1 means it points to next location as it is address and it depends on size of the data type i.e 4 bytes so 104 will be stored in next p[1] location.

*(p+i) here means *(4058+1) ⇒ *(4062)

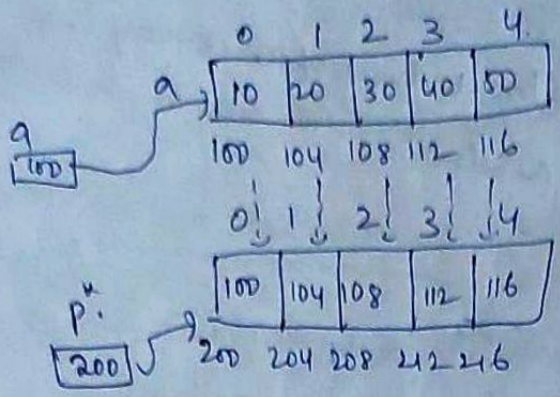
at this 4062 address we store 104. ... so on.

So This is how we are creating array of pointers, which stores references into those locations.

How we are accessing the information or values through p:

How to read the elements of array of pointers?

Suppose, we are taking 2 arrays:



// one array is storing integer values.

// one array is storing reference of every integer element of the above array.

p is an integer pointer.

Now how can we process or simply how can we display the elements.

We have two options:

a) either we go with the index

b) either we go with pointer modifier.

With the help of index we will access the elements:

```
for(i=0; i<5; i++)
{
  printf("%d", a[i]); // using index and using array directly.
} printf("%d", *(a+i)); // not pointer to array.
```

↳ we can access like this also.

*(a+i) ⇒ means a holds Base address i.e. 100

and initially i=0 and a+i ⇒ means 100+0 ⇒ means 100 and *(100) means value at 100 = 10

and i will be incremented to 1

now *(a+i) ⇒ *(100+i) ⇒ *(104) = 20 — so on.

This is how we are processing the elements directly by using the name a.

Accessing the elements using array of pointers:

```
for(i=0; i<5; i++)
{
  printf("%d\n", *p[i]);
}
```

p[i] ⇒ p[0] which we have 100
and *(100) = 10 so on...

If i dont want to use index, i want to print using pointer notation,

```
for(i=0; i<5; i++)
{
  printf("%d\n", **(p+i));
}
```

**(p+i) ⇒ 200 + 0 ⇒ 200 **(200)
⇒ *(*(200))
⇒ *(100) = 10.

Better to use pointer notation only, Whenever we go with the dynamic memory allocation concepts, dynamic programming always we use pointer notation approach.

In case of static programming, we use the indexing approach.

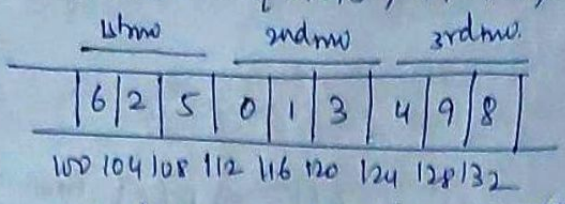
⇒ pointers and 2d arrays.

contents:
- How 2d arrays are related with pointers.
- How you can access the elements of the 2d array with the help of using pointers rather than using array name.

→ We have discussed the relationship between 1d array and pointers.

Ex1

int a[3][3] = {6,2,5, 0,1,3, 4,9,8};



actual representation of 2d array in memory

Logical representation

	0	1	2
0	6	2	5
1	0	1	3
2	4	9	8

3 rows X 3 columns 3 X 3 Matrix

The above is the way the 2d array is declared & initialized at compiletime

We have declared and initialized the 2d array at the compiletime itself).

→ We have considered here row major implementation of 2d array. first row will be stored then second row will be stored and then third row will be stored in memory using row-major implementation.

→ A 2d-array is an array of arrays.

→ If you see the logical representation of array above:

we have 3 arrays.

6 | 2 | 5 → is 1d array.

0 | 1 | 3 → is 1d array.

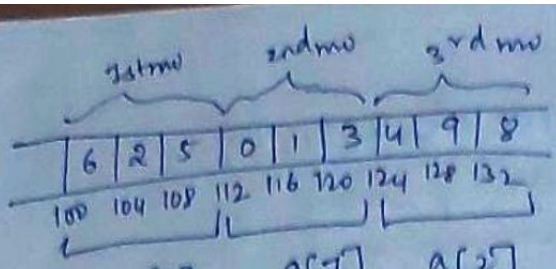
4 | 9 | 8 → is 1d array.

} each 1d array is having 3 integer elements.

and if you combine the above 3 1d arrays, then it forms a 2d array, so you can say that 2d array is what?

It is an array of arrays.

→ The above 2d array is an array of 3 1d arrays. and each 1d array is having 3 elements



this is one 1d array

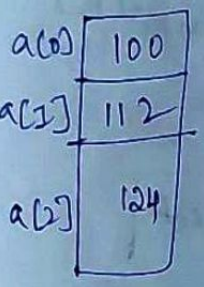
this is another 1d array

this is another 1d array.

⇒ So a 2d-array name is having 3 1d arrays. (a)

a[0], a[1] and a[2] and each 1d array is having 3 elements of integer values.

How you can write this a[0]'s.



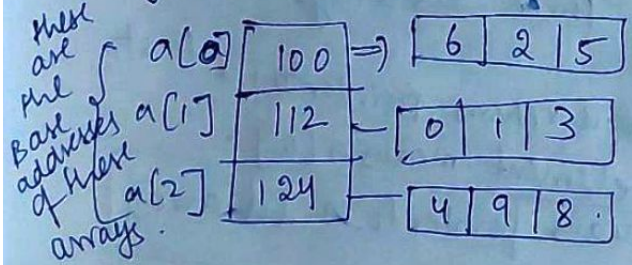
first element is a[0], if you simply write a[0] it means, it itself is a 1d array and the name of the array contains what? address of its first element of the array. or you can say it points to the first element of the array.

It means if you say a[0] ⇒ it means here 100 (the element of its first address)

If you write a[1] it means it itself is a 1d array and if you say a[1] it means it is the name of the array, it contains base address or the address of the first integer element of array. i.e 112

Similarly a[2] contains 124.

You can write like below.



⇒ a[0] contains 6, 2, 5 values

⇒ a[1] contains 0, 1, 3 values.

a[2] contains 4, 9, 8 values.

i.e. why 2d array is array of arrays

Ex: int a[3][3] = {6,2,5,0,1,3,4,9,8};

Now let us take a pointer.

int *p;

/* This pointer is going to contain address of another variable and the datatype of that variable is integer.

So it is a pointer to integer. */

VV IMP. p = a; ⇒ can we write here a (name of the array) as we have discussed in 1d array?

In 1d array, it is fine, but here it is not right it is invalid.

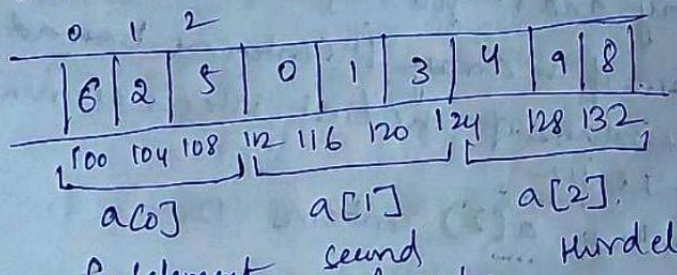
Why so -)

⇒ if you write this single like a, a is the name of the 2d array, and what the array name returns -)

array name returns the pointer to its first element.

⇒ In 2d array the first element is what)

In this 2d array we are having 3 elements



you can say it is having three 1d arrays.

So if you write p=a; it will return pointer to its first element. means a[0].

VVV IMP means pointer to a 1d array.

vvimp.

It is not going to return pointer to an integer

But p can contain address of an integer variable only.

So `p=a;` is invalid in 2d arrays.

You can write like below:

Ex: `int a[3][3]={6,2,5,0,1,3,4,9,8};`

`int *p;`

`p=&a[0][0];`

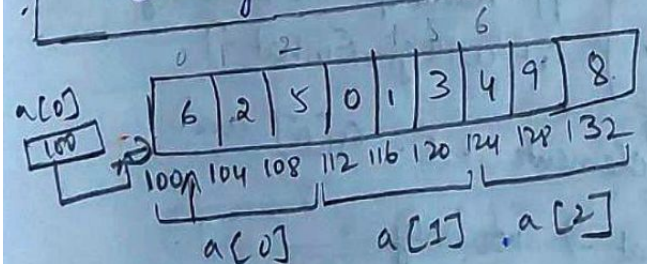
a[0][0] means value 6 and address of this is 100

p returns &a[0][0] ⇒ means it will return the address of an integer variable. (6 is an integer variable and it is going to return address of 6 (integer variable).

(or) we can write `p=a[0];` this is also valid now

Why so →

If you write a[0] ⇒ It means the 1st 1d array and if you write the name of this array (i.e [a[0] here), the name of the array always returns pointer to its first variable. or you can say address of its first variable.



here you can see a[0] contains base address of this 1d array i.e 100 so a[0] is name of array and it points to first element of a[0] → 1d array.

So if you write `p=a[0];` ⇒ 100 will be stored at this pointer (p). as pointer can contain address of another integer variable.

Note:

But we cannot write `p = a[0][0];` because a[0][0] will return value i.e. 6, but pointer cannot contain a value. So this is invalid.

Printing the address of 2d array:

If you want to print the address of 2d array, the base address i.e. 100 here

If you want to print this address in hexadecimal form, you can write down %p, or %u, it will print in unsigned int

vvimp

```
printf("%u", p);
```
// 100

If you simply print p, it will return what? We have declared this pointer p somewhere in the memory and pointer p is allocated some memory location.

So p = &a[0][0]; ⇒ i.e. it is going to store 100.

So now p points to 6 element / 100 address.

vvimp ⇒ Rather than p we can write simply a (name of the array).

```
printf("%u", a);
```
// 100

name of the 2d array it will give what? The address of the first element i.e. a[0] and address of a[0] is what? 100 only.

```
printf("%u", &a[0][0]);
```

`printf("%d", &a);` // another way to print address of 2d array

&a → a means complete array name then &a means it points to the Base address of Whole array
100

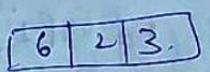
`printf("%u", *a);` // another way to print address of 2d array.

⇒ Now *a will print 100, because * means dereferencing operator, it will print the value at that address.
100

→ If you write simple a → it will return what a is the array name, it will return that pointer to its first element"
or you can say that address of its first element.

→ In the 2d array, the first element in this a[0] which is a Complete 1d array because 2d array is having 3 elements
i.e three 1d array's.

⇒ So a[0], and *a[0] means value at a[0] means the Complete 1d array i.e

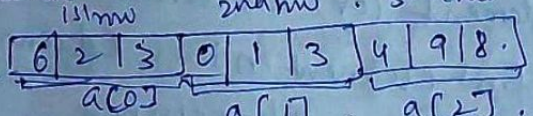


→ value doesn't mean here it will return 6, 2, or 3 here, because at a[0] it will return the address element of the complete 1d array, so it will return the Base address of this 1d array i.e 100

a[0] = 100

⇒ `printf("%u", a[0]);` // this is another way to print address of 2d array.
100

Because a[0] is the name of the the Complete 1d array and if you write the name of the array it will return the base address of the first element in this array. (i.e 6 elements address) so it returns 100.



Very Important point in 2d arrays.

> Although this a and &a is returning the same value but they are different

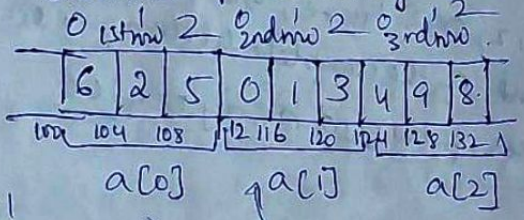
⇒ Now if you print (a+1) like below:

```
printf("%d", a+1);
```

112

WW INP

a ⇒ is the name of the 2d array, it will return what? pointer to the first element in this 2d array and first element is a[0] (complete 1d array) and a[0] is a pointer to this 1d array and if you add 1 to a pointer, it is going to point to the next element so it will point to a[1] ⇒ (next 1d array or next element of 2d array.



a+1 means ⇒ it points to a[1].

and now it is pointing to a[1] and if you print a+1 then it will return the base address of this 1d array's first element (or) we can say it points to first element in this 1d array (a[1]) i.e. 112 will be returned.

⇒

printf("%d", *a[1]); // because a[I] you can write as *a[I] also

both a[I] and *a[I] will return the base address of the array i.e 112 will be printed.

// In 1d array a = *a. and here a[I] is 1d array. so we can write a[I] = *a[I]

printf("%d", *(a+1)); // here also it will return 112

here a means name of the 2d array and it is going to point to the first element i.e. a[0] or we can say it is going to return the address of first element i.e. a[0]'s address and it is 100 and if you add +1 means it points to next element i.e. a[I] which is 112 (or) a[I] is again a 1d array and i.e. now it points to a[I] address

in a[I] the first element is 0 and its address is 112 and as a[I] is an array name it will return the Base address of (1d) pointer to 1st element of this a[I]. So returns 112.

printf("%d", a[1]); // this also prints 112

*(a+1) ⇒ means it will return complete 1d array so its address is 112

printf("%d", *a[1][0]); 112

⇒ Suppose you print:

printf("%d", *(a+1)+2);

*(a+1)+2 ⇒ a is going to return. What? i.e a[0], address of first element and first element is this complete array and if you add +1 to this

(1d)

6	2	5

complete array, it is going to point to next element in this array, so next element is also going to point to a[I] and it is also a complete 1d array.

and *(a+1) means value is complete 1d array, so it returns the base address of this complete element i.e. 112

and in this 112 if you add +2 ⇒ it means it is going to add how many bytes (depends on datatype) here it is integer and integer is going to take 4 bytes so +2 means it will add 8 bytes.

so it will add 8 to 112 i.e. 120 so

WAY IMP

*(a+1)+2 will return 120

print the value at 120th location.

We have to use de referencing operator (*).

printf ("%d", *(*(a+1)+2)); =3.

((a+1)+2) ⇒ *(120) = 3.

printf ("%d", a[1][2]); =3

row col

Suppose you want to print 4 in array:

printf ("%d", a[2][0]);

printf ("%d", *(*(a+2)+0)); 4

rowno colno

General formula to access the value of 2d array i.e a[i][j]

$$a[i][j] = *(*(a+i)+j)$$

If you want to write this a[i][j] in the form of pointer, then how you can write-?

$$*(*(a+i)+j) = a[i][j]$$ ⇒ both are same.

If you want to use the pointer then:

pointer is also going to store the base address. Like

```
int a[3][3] = {6,2,5,0,1,3,4,9,8};
int *p;
p = a[0]; // p = &a[0][0];
```

p is a pointer which we are pointing to 2d array and it stores the base address of complete 2d array.

So rather than using a u can write pointer name here.

$$a[i][j] = *(*(p+i)+j)$$

Rather than $*(*(a+i)+j)$ or $*(*(a+1)+2)$ How can we write this is:

we can see: $*(a+i)$ = (is equal to) $a[i]$

so $*(*(a+1)+2)$ can be written as $*(a[1]+2)$

so $a[i][j] = *(p[i]+j)$ $a[i][j] = *(a[i]+j)$

So we can access the elements of 2d array using:

a) a[i][j]
b) *(*(a+i)+j)
c) *(a[i]+j)

} in all these ways.

d) *(*(p+i)+j)
e) *(p[i]+j)

} here p is a pointer, so using pointer we can access the elements of 2d array like this.

If you want to print:

*(*a+1) =?

Step by step explanation:

*a

⇒ a means it is going to return what-?

a is the name of the 2d array, so it is going to return the pointer to its first element" or we can say "base address" (or) "address of its first element".

⇒ first element in 2d array is full complete 1d array (a[0])

(taking the same above 2d array example).

so it is going to return pointer to the 1d array i.e. pointer to a[0] i.e. 100.

⇒ *a means value at a i.e. value at complete 1d array or we can say the address of the first element of this 1d array i.e. 100 it will return and value at a[0]=100 (pointer to its first element).

Now 100+1

100 is a pointer, if you want to add some arithmetic or some integer value, it is going to point to the next element and it will add 4 bytes (depends on datatype). so *a+1 = 104.

→ *(*a+1)

*(104) so value at 104 = 2. so it returns 2.

If you print:

**a =? What will be the output?

**p =? What will be the output?

a[1]+1 =?

*a[1]+1 =?

a[1]+1:

a[1] means it is 1d array (2nd element in 2d array) and name of the 1d array is a[1], and name of the array returns the pointer to its first element / base address of the array i.e 112 so a[1] will return 112

and if you add +1 ⇒ it is going to point to the next value i.e 112+4 = 116 points to 1 element.

So a[1]+1 ⇒ 116 (returns 116).

*a[1]+1:

a[1] and *a[1] both will return 112

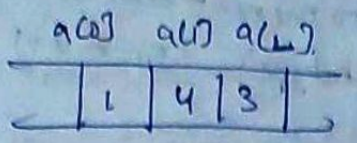
So you will say a[1]+1 and *a[1]+1 will return

116 —? No it is not right.

How these are different, we will discuss with 1d array.

Suppose we have taken 1d array of size 3.

```
int a[3];
```



if you print a and &a

a means name of the array and it contains BA = 200

&a ⇒ means address of a, both will return 200 (200)

a+1 = } will show same NO.
&a+1 = }

⇒ a ⇒ In this 'a', simply we are writing the name of the array (means it will return pointer to its first element)

address of the first element i.e 200

$\boxed{a = \&a}$ ✓

⇒ &a ⇒ address of a means it is going to give you the address - or you can say the base address of the complete array. So this will also give 200

$\boxed{a+1}$.

but if you add +1 to both, in this case it is different.

a+1 ⇒ a means it is pointing to 1st element i.e 200 and if you do +1 (arithmetic on pointers) then it is going to point to the next element i.e 204

Ka+1

↳ means: 200 + 12 ⇒ 212

Ka → is going to give you the pointer to the complete array. i.e also 200

But if you do "+1" it means it is going to add how many bytes-?

Size of the array is 3, so 3 × 4 i.e 12 bytes.

So it is going to add 12 bytes

It is going to give the address of the complete array.

i.e. it returns 212 and it points to 212 element.

Same in this case! (In 2d array)

a[i]+1 = 112 + 4 = 116

Ka[1]+1 = 112 + (Size of array = 3 · 4)

= 112 + 12 = 124 ✓

So Ka[1]+1 will now point from 112 to 124 now.

i.e it points to the next 1d array now.

This is the difference between a[i]+1 and Ka[1]+1.

P-T-O

⇒ Passing an array to a function:

→ When we were passing an arguments in function either we have passed int variable, float variable, single character variable as an argument.

→ But what if you want to pass complete array as an argument, string as an argument, pointer as an argument to the function.

How to pass complete array as an argument in function.

- Why we pass a complete array
- What is the need to pass an array as an argument.
- What is the mechanism behind passing an array as an argument...

⇒ How to pass an integer/float/char as argument we know.

Suppose lets take an ex:

```
void fun(int);
void main()
{
    int x=5;
    fun(x);
}
// function definition
void fun(int a)
{
}
```

Why we pass array-?

→ suppose in a class there are 60 students, and i want to create a function to calculate the average of marks of 60 students so I want to pass marks of 60 students in that function.

→ A function is like:

avg (; pass marks of 60 students here).

→ One method is what here you can pass 60 variables and individual variable is containing marks of 1 student. we also way but that is not a good way! practice.

→ We are not going to pass 60 variables / 60 variables here

→ Rather than this, to store same type of data rather than passing 60 variables, we use array.

Instead of passing 60 variables we can pass array[60]

Students marks.

→ int marks[60]; // we took int marks[60]; we can simply pass this array. rather than passing 60 variables and this is good practice.

→ So when you pass list of values to a function then we use array.

→ This is also useful when you pass string to a function.

That is one great application of passing array as an argument (string as an argument).

How you will pass array as an argument:

We shall discuss with a complete program.

First of all we declare a function and i want to calculate the average of 5 students

```
int avg(int[]);  // declaration of avg function
                 // and as a argument
// i am expecting it would pass an array and in
   declaration no need to write down the name of that
   variable whatever you pass.
```

You have to tell only that it should be of integer type and this would be array (we just write [] Subscripts.

```
Void main()
{
  // In mains function I declare an array as
  // below  int average;
     int marks[5] = {10,15,20,30,45};
        // Now I want to pass these array.
  // How we will call this function as below.
  average = avg(marks);  // name of the function (name
  printf("average is %d", average);  of the array).
}
```

In function definition what should I do, I want that it should return average.

// function definition. // datatype named (datatype array function name [])

int avg(int marks[], int a) → this as an argument.
no need to specify the size

// when you pass array as an argument then in definition data type arrayname[] → would be the case and in function calling only pass the name of the array.
and the declaration is datatype functionname(datatype[]);

no need to specify the size then

{ // first we will calculate the sum and then avg = sum divided by the no of students. and we don't know how many no of students or we don't know the size of the array here, so size also you want to pass, then how you will pass the size of the array. In function calling you can call like 'average = avg(marks, 5);' or if you don't know, you can calculate the size of the array. (will discuss)

```
int i, sum=0, average=0;
for(i=0; i<a; i++)
  { sum=sum+ marks[i];
  }
  average = sum/a;
  return (average);
}
```

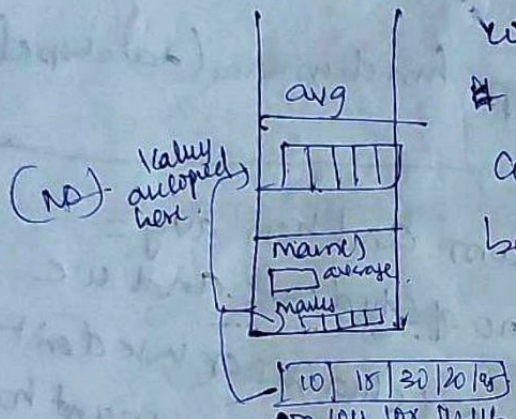
This is how you will pass an array as an argument.

Explanation of how the array is passed:

When you execute the program the control goes to the main() function.

So in stack, the memory will be allocated.

Suppose i take the following as a stack.



whenever we run the above program, we will go to main memory, for some coding that thing, memory will be allocated and for static / global then a stack memory / heap memory

So in stack, for the main() the memory will be allocated. (it is like one frame or it is like one activation record for the main() function will be held).

and in this memory will be allocated for average and for marks array and in this we have 5 elements and size of datatype int so 5 x 4 = 20 bytes will be allocated for this marks array.

⇒ name of the array will act as the internal pointer (constant internal pointer) and it contains the base address of the array. (100).

marks [100] ⇒ it acts as internal pointer to the first element.

Next line in main function is calling of the function avg.

~~avg~~ `average = avg(marks, 5);` → This is calling of function avg

and we are passing marks, (name of the array) whenever you will write down the name of the array, the function is called so it goes to the avg function definition and for this function also the memory will be allocated and in that we within we have another array marks[] which is inside avg function (u can take different name also for this function marks in function definition and the values of marks which are declared in main function will be copied to the function avg and in that marks array.

→ and in 'a' variable, i.e size, 5 will be copied.

⇒ But the above is not the mechanism behind this.

When you pass the array (actual mechanism)

It is going to pass the base address of the array.

So when you call

average = avg(marks, 5);

↳ marks means name of the array and it contains the base address so it will pass only 100

So in function definition int marks[] will act as a pointer.

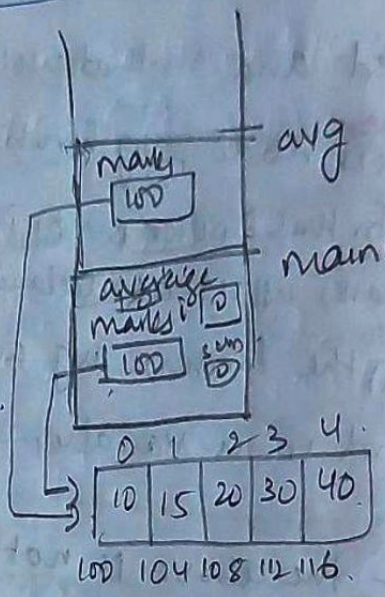
How the Compiler will interpret this line?

int marks[] ⇒ int *marks;

means it is a pointer / name of the pointer is marks, and it is pointing to a integer value, integer type pointer.

> Not the complete values will be copied.

Here in avg function in marks (pointer) we will have 100.
So marks in function points to marks array we points to the first element of this array.



→ We are passing address so this is an example of call by reference. not call by value.

Very imp point

> so by default array would be passed by call by reference method not call by value method.

It is right that array is called by reference because:

Suppose array is having marks of 100 students or a list of values are 200, so when you pass 200 values now a separate copy will be created here and 200 values will be stored. if you pass it by call by value. that is of no use.

It is wastage of memory.

In main function where you declare 200 values and in function definition in the copy 200 values and again you call this function again it would be in memory and again 200 values will be allocated. Loops will be stored.

i.e any array will always be passed by Call by reference

VVIMP.

So in function definition int marks[] will be acted as reference like int *marks, it will not act like a complete array, it is not a duplicate array, int a this is what we are passing is call by value,

the for loop: i=0, sum=0, average=0

i=0; i<5 true enter the loop.

and sum = 0 + marks[0]; ⇒ 0+10=10

↳ means it will access the element at 0th index and value is 10.

in sum = 10 now and again i++ so i becomes 1, and it will access the next value = 15 so 10+15=25 and the process continues and totalsum = 120.

avg = 120/5 = 24.

avg value = 24 and will return 24 to main() and whatever is returned will be stored in average and prints average.

// If you want to know the marks is acting as reference / value.

in main() function and avg function type.

printf("%d", sizeof(marks)); in main it will print 20 and in avg it will print 4.

// Calculating the size of array in bytes.

Size is 20 bytes and divided by 5, size of single value i.e a[0] = 4 bytes.

```
#include<stdio.h>
int avg(int [], int);
Void main()
{   int average;
    int marks[5] = {10,20,30,40,50}; size;

    size = sizeof(marks) / sizeof marks[0]);

    average = avg(marks, size);
    printf("average = %d \n", average);
}   Printf("Inside main sizeof array(in bytes) is: %d", sizeof(
int avg(int marks1[], int size1)
{
    int i, sum, average;
    for(i=0; i<size; i++)
    {
        sum = sum + marks1[i];
    }
    average = sum / size1;
      Printf("Inside avg fn return size of array: %d",
    return (average);                    sizeof(marks1));
}
```

(returning pointer from function)

contents → How to return a pointer from a function.
→ returning an integer/ float Pointer from a function.

→ We can also pass a pointer as an argument to a function.
ie, nothing but call by reference method.

Example:
- We will take a function named return pointer.
- We will take an array which we will pass in the function and then in this function, I will increment the address of that pointer that ur passing by 2 or 4. and then we are going to return that address.

Program:

```
int* returnPointer(int []); // What would be the return type is important here.
```

We are returning a pointer so what will be the return type
We are taking integer typed array as an argument. So return type will be int*

If you write int → it will return integer value.
We want the function to return a pointer.

So int* ⇒ integer pointer.

```
int * returnPointer(int []); // This function is going to accept
```

array as an argument and it is going to return integer pointer or pointer to integer.

In main() we call the function, and it is returning a pointer means we have to accept the value whatever this function is returning and it is returning a pointer type value (integer pointer).

```
void main()        int * returnpointer(int []);
{ int *p; // To return a pointer we are taking
                  int *p.
  int a[] = {1,2,3,4,5}; // took a array and initialized
  p = returnpointer(a); // calling the function and passing
  printf("%d", *p);      // array as an argument and
}
// in order to pass the arrayname as argument we will pass just
// arrayname (a hex) and whatever this is going to
// return i will store it in p. p is an integer pointer.

//Definition of function.
int * returnpointer(int a[])
{                    ↳ Compiler will interpret this line as
                       *a. means here a is a pointer. which will
  a = a+2; // updating a.   store the address of the first
  return a;                  element.
}
```

Explanation:

Whenever you execute the above program, the control goes to the main function, if you don't mention any datatype for main(), by default it will consider int, if you write void it will not return anything.

→ So in main we have declared a pointer variable
So Generally the memory which will be allocated to this program, will be divided into 4 parts.
- Some memory to store the instructions we are going to return
- Some memory for static and global variables
→ one part of memory is stack memory and one memory is heap memory.
→ so whatever we are declaring and array declaration

initialization and calling function & printing in main() function all will be stored in stack (memory of this function).

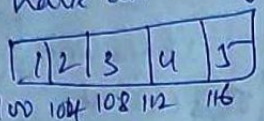
stack memory of this function.

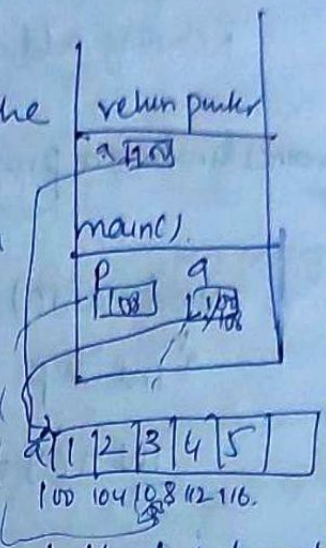
→ The main() function for the memory has been allocated to this function.

from stack, one stack frame will be for the main() function.

→ In this we have a pointer p for which memory is allocated

→ we have an array and initialized





→ In a we have 100, and a is pointing to the first elements address.

→ a is an internal constant pointer to the array.

→ In the next line we are calling "return pointer" function, now the control goes to return pointer function definition, and for this function also one stack frame will be allocated.

→ In a[] nothing but a pointer. and here what we are passing→ we are passing a (i.e array means we are passing (100).

⇒ a=a+2 → means we are incrementing the pointer.

a+2 → it does mean 100 + 2 = it points to 3rd index and 3rd position element address. (not 102) (100+2×4) = 100+8 = 108)

now a contains 108 and it is pointing to 3rd element now

So it returns 108, to main() function control goes to and 108 is address and to store this value we need a pointer, so return a' means 108 is returned and stored in p.

→ so in p also we have 108. p points to 108.

*108 value at 108 is 3 so it prints 3.

Returning a pointer means we are returning address.

Obviously we have returned address, in 'a' we have address so.
We can say we are returning address we are not returning pointer.
Returning address means returning a pointer. (same thing)

In main() function a program change!

```
void main()
{ int *p;
  int a[] = {1,2,3,4,5};
  a = a+2;
  printf("%d", *a);
  p = returnpointer(a);
  printf("%d", *p);
}
int *returnpointer(int *a[])
{ a = a+2;
  return(a);
}
```

If you write like this it will give error. a is constant pointer to the first element of the array we cannot do increment decrement with 'a'.

// here it will not give error because it is not an array, we are writing this as an array, actually the compiler interprets this as pointer (just a pointer).
int *a; (formal pointer)
and we can do pointer arithmetic operation.
p++, p--, p+2 any

Memory allocation functions

Dynamic memory allocation:

- Dynamic memory allocation - means allocating the memory for the variables at runtime.
- i.e. Allocation will be done at runtime.

Let us consider the arrays concept:

We have to declare the array means we have to specify the size of the array while declaration.

int marks[10]; ⇒ this implies marks is an array of integer datatype and it can hold maximum of 10 elements.

10 → maximum size here.

→ we may use the 10 elements or we may not use the 10 elements.

Then are 2 cases:

First case. a) If we are using only 5 elements suppose.

the remaining 5 elements, memory will be wasted.

Suppose int occupies some 2 bytes of memory / 4 bytes depends on the compiler

so total memory occupied by the marks array will be 10×4=40 bytes or 10×2 = 20 bytes of memory (depends on type of compiler).

and we are using only 5 elements × 4 ⇒ 20 bytes of memory.

(or) 5 " × 2 ⇒ 10 bytes of memory.

So remaining 20 bytes | 10 bytes will be wasted.

Without declaring the size array declaration is invalid.

Case 2:

10 elements are there → and each occupy 4 bytes i.e. 40 bytes

Suppose you want to add 3 more elements now.

This cannot be possible, even though the array is having limitations on boundaries but because we have given / declared size at the compile time the above 2 cases we face.

The above is called compile time memory allocation / static memory allocation.

To avoid / for avoiding the above cases, we can allocate the memory at run time (dynamic memory allocation.

At run time:

⇒ At run time we need not go with arrays declaration, at the run time whatever the required memory we need that memory can be added / allocated.

For this dynamic allocation of memory / run time allocation of memory we have three functions.

a) malloc function.
b) calloc function.
c) realloc() function (increasing the memory).
d) free() (deallocation of memory).

Wrong

⇒ As we are allocating the memory during run time, so the user is responsible to deallocate the memory after its use

Unless the user deallocate the memory, the memory will not be erased.

⇒ If you are allocating the memory using one among the above three functions, it is required to free that memory using free function.

malloc() function:

malloc() ⇒ stands for "memory allocation"

So whatever the function we are using for allocating the memory that will be in a pointer type.

It is a void pointer or returns a void pointer, so that we can typecast to our required datatype.

Syntax for malloc() function:

pointer = (typecast) malloc (size).

So if we want to allocate the memory:

Ex: Suppose I want to store integer array.

ptr = (int *) malloc (n * sizeof(int)).

(int * ⇒ typecasting to int *; n ⇒ want to store integer values in array.)

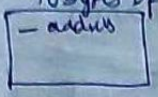
⇒ sizeof ⇒ will give the size of the datatype.

⇒ The above declaration will be allocating, if n = 5, then sizeof int (2/4 bytes depends on compiler, 16 bit - 2 bytes) then 5 x 2 = 10 bytes / 5 x 4 = 20 bytes will be allocated to ptr (pointer) and this pointer will be addressing at beginning.

actually ptr is a void pointer, so we have to typecast in the required format and then malloc function and you have to specify the size.

⇒ In the size if you want to store some 5 variables or 6 values that will be mentioned here as "n * size(int)"

→ If it is a float, we can give the sizeof float then 4 bytes of memory will be occupied for each float value in a 16 bit compiler (4×5 = 20 bytes will be allocated) to the ptr.

⇒ and 

and this ptr will point to the beginning address / base address / starting address.

→ for each and every value we have to increment the pointer, so this (p+1). It is an integer, so 2 bytes of memory will be occupied for each element

So this is how we can use this malloc function.

So the complete 10 bytes of memory will be allocated to the pointer

calloc function:

calloc function ⇒ means contiguous memory allocation.

⇒ Here the memory will be allocated in terms of blocks.

Syntax:

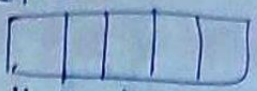
Same syntax, here also it will return a void pointer so that we have to typecast to the required format.

```
ptr = (typecast) calloc (#Number of blocks, size);
```

Ex: Let us take the previous example (for malloc we took) We have to store 5 values, so i am giving

```
ptr = (int *) calloc (5, sizeof(int));
↓
void ptr.
```

here 5 blocks will be created ptr is pointing here.



each of 4 bytes.

4 4 4 4 4 bytes

And pointer points to starting address.

→ So if you increment the pointer automatically it will point to the next memory location.

Suppose float:

```
ptr = (float *) calloc ( 5, sizeof (float));
```

typecasting to float pointer.

- ptr pointing to starting address of memory.

In Malloc function;

ptr = (float*) malloc (5 * sizeof (float)). Complete block of 20 bytes will be allocated to ptr and ptr points to starting address.

Where as calloc function - memory is allocated in blocks.

Only difference between malloc() function and calloc() function is
In malloc() function we use only one parameter and in calloc()
function we use 2 parameters.

The above two are used to allocate memory at runtime.

These will be used in data structures like linkedlists --- so on.
We go with dynamic memory allocation in these.

p-7-0

realloc() function:

- If the memory allocated with the help of either malloc() function or calloc() function is not sufficient and if you want to add more memory then you will go for realloc() that means re-allocation.

> realloc() ⇒ stands for re-allocation of memory

i.e. We can increase the existing memory which is created with the help of malloc() or calloc() functions.

Syntax:

ptr = realloc(ptr, newsize)

ptr ⇒ Increasing the pointer with newsize and this pointer will be increased.

newsize ⇒ How much we need to increase we have to mention here.

Ex:

ptr = (int *) malloc(5 * sizeof(int)) ⇒ this statement will allocate 20 bytes of memory to ptr.

Now i want to reallocate the memory for above:

ptr = realloc(ptr, 4 * sizeof(int)); 20 + 16 = 36

ptr = realloc(ptr, 10); 20 + 10 = 30 bytes.

> For compile time initialization we cannot use realloc()

free();

> This memory is allocated by the user at runtime, so it is necessary to release the memory / deallocate the memory by the user itself.

realloc() function:

- If the memory allocated with the help of either malloc() function or calloc() function is not sufficient and if you want to add more memory then you will go for realloc() that means re-allocation.

> realloc() ⇒ Stands for re-allocation of memory

i.e. We can increase the existing memory which is created with the help of malloc() or calloc() functions.

Syntax:

ptr = realloc (ptr, newsize)

↓ Increasing the pointer with newsize and this pointer will be increased.

↓ How much we need to increase we have to mention here.

Ex:

ptr = (int *) malloc (5 * sizeof(int)) ⇒ this statement will allocate 20 bytes of memory to ptr.

Now i want to reallocate the memory for above:

ptr = realloc (ptr, 4 * sizeof(int)); 20 + 16 = 36

ptr = realloc (ptr, 10); 20 + 10 = 30 bytes.

> For compile time initialization we cannot use realloc()

free():

> This memory is allocated by the user at runtime, so it is necessary to release the memory / deallocate the memory by the user itself.

for deallocation purpose we are having free() function.

Unless you use this free() function, the memory allocated by using the malloc(), calloc() (re alloc() will not be released.

Syntax of free():

```
free(ptr);
```

// Whatever the pointer we are using during the allocation, that pointer should be given as an argument for this free function so that whatever the memory we are allocating using malloc() (or) calloc() (or) realloc(). That memory gets released.

Imp

It is the responsibility of the user to release the memory after using the memory.

These functions are used in data structures (Linked lists) to save the memory (No wastage of memory.

→ No chance of getting out of bound exceptions if use these functions

Pointer to a function / function pointer:

→ How to create pointers to functions.

→ What is a pointer to a function.

→ Generally we know how to create pointers to primitive data types such as character, integer, float, double.

⇒ It is possible to create pointers to functions also

↳ But if you want to create a pointer to a function, the declaration depends completely on the function prototype.

→ The declaration of the function depends on the prototype of the function.

Syntax:

Declaration:

```
return_type (* Identity)(args_list)
            (pointername/Identity)
```

↓ pointername

↓ how many arguments it is going to take.

↳ This is declaration / syntax of pointer to function.

⇒ Just Identity → you must place inside the parenthesis along with the *(pointer).

return-type and the arguments list.

Let us understand with one example:

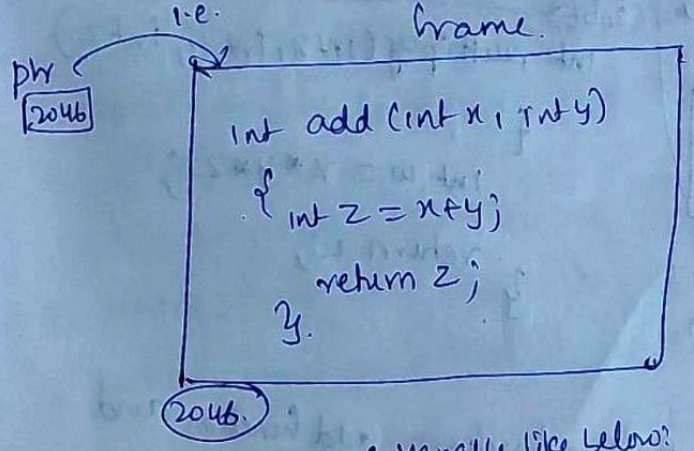
Functional pointer / function pointer:

We will take two functions:

```
int add (int x, int y) // we took add function which is
{                      // taking 2 arguments of int and
  int z = x+y;         // performs addn, returns z of type int
  return z;
}
```

Here the function is ready.

⇒ Now we have to create a pointer to the above function completely.



→ to this complete function we have to create a pointer.

We have to write a variable like below:

```
int (*ptr)(int, int)
```

- int → depends on the prototype of function. Here add function have 2 arguments and their datatypes are int. and return type of these functions is int.
- (*ptr) → place it in parentheses. identify. any name we can give.
- (int, int) → arguments

So now our pointer variable / function pointer is ready.

Plus
⇒ *ptr → can point to any function which is taking two integer arguments and returns int data (not only add it is we took for ex).

Viva
⇒ Such type of functions, any functions that's pointer i.e. can points to, this is (return type) int (*ptr)(int, int)
called pointer to function declaration.

Program illustrating pointer to function:

```
main()
{
  int r1, r2, r3;
  int (*ptr)(int, int);
  r1 = add(10,20);
  r2 = multiply(2,3,4);
  printf("%d %d", r1, r2);
  ptr = &add;
  r3 = ptr(30,50);
  printf("%d", r3);
}
```

direct calling

calling with ptr.

We are storing in r1 again.

ptr 2046

Frame

```
int add(int x, int y)
{
  int z = x+y;
  return z;
}
```

2046

```
int multiply(int x, int y, int z)
{
  int w = x*y*z;
  return w;
}
```

Explanation:= We have took two functions, add function and multiply function.

→ add function is taking two integer arguments & its return type is integer.

→ multiply function is taking 3 integer arguments & its return types is integer.

→ Based on the above two functions, we do write main() function.

→ We have declared 2 variables r1, r2 and we are calling add function (direct calling) we are passing directly the values, and storing in r1.

→ We are calling multiply function and we are passing 3 integers i.e. 2,3,4 and the result we are storing in r2.

→ We print r1, r2 values i.e 30 and 24 respectively. (10+20) (2×3×4)

⇒ This is direct calling.

Now the Concept is:

Creating a pointers to the above functions and accessing.

I will declare only one functional pointer / one pointer, it is a local variable so declare after names function (here).

Like below:

int (*ptr)(int, int); ⇒ I have declared like this in main() function → Now this can point to only add function and it cannot point to multiplication function

→ The reason is the prototype is different.

Now we have stored "address of add function" in the ptr variable because "ptr" → is a pointer variable and it holds address.

→ Whenever a frame will be created for the function / method Space will be created for a function, the frame will be created at a particular location suppose 2046 → this is called as the entry point of the function execution.

With ptr → we are storing that entry point to point to that function / access the function.

// So whenever the add function address we are assignin
that means the frame address (memory will be
allocated inside the RAM to execute that function (
inside the stack memory) this address we are calleu
pointer (ptr).

// If you want to call instead of using address (&add) I use u
ptr (30,50) => which will point to frame in which a
function is written.
and we are storing in r1 again, because working u
r1 is over. or you can use r3 (another value).

Note:

```
ptr = &multiply;
```

But if you store address of mu
to ptr it will give error
because prototype doesn't match
and you will get error message, incompatible pointer
assignment (list of arguments don't match).

Why we have to put *ptr in parenthesis -) i.e why
we are putting pointer variable identity inside the(parenthesis)

-) Declaration:

int (*ptr) (int, int).

If we re-write as

```
int *ptr (int, int)
```

-) we removed parenthesis
here.

If you remove the parenthesis what will happen -),
then system / compiler considers *ptr as
int *ptr(int, int)
↳ here ptr is not a pointer, but it is a function
which is taking 2 integers and its return type is pointer (K integer variable).

int (*ptr) (int, int) → here ptr is a pointer which is pointing to a function which is taking two integer arguments and it is returning integer type.

So the complete meaning gets changed if you don't put the identity of pointer in parentheses.

Strings in C:

Contents:

Basics of strings
→ What is a string
→ How string is different from a character array
→ How to declare a string and initialize the string
→ different methods of initializing a string with examples.

Strings:

What is a string?

In simple terms if you say in C.

"string is simply array of characters"

General syntax of array declaration:

datatype arrayname[size];

array means collection of more than 1 data items that are of same type. (either int or float or char).

→ string is an array of characters. means how to declare a string

datatype must always be a character in a string.
We can only store characters.

Declaration of a string → char stringname[size];

We cannot write any int/float or anything at all.

only difference between array and string is, in array we can take any datatype int, float, char, but in string you can only take character datatype, because string is an array of characters or we can say it as an character array.

We can say a string is character array, the only difference is what it is:

> string is a character array which is ending with null character.

Justification:

```
Char name[10];
```

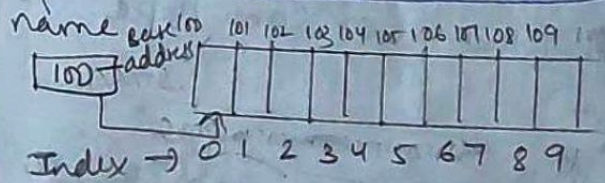
This is the declaration of string and name of the array is "name" (here).

Size is 10.

→ string will always be ended with a null character.

string array name → name

How the memory will be allocated?



10 bytes will be allocated to the array (name) ↓ here it is name of the array.

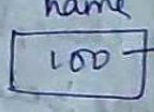
because the character size is 1 byte.

→ suppose address is 100

Next byte will be 101 then 102 then 103 --- till 109.

→ here name of the array is name (String name).

→ and name is containing what the base address of the string.



→ It is address not value.

→ array is considered as an internal pointer because name of the array variable contains base address of the array or address of the first element of the array

and string is an array of characters, obviously it is an array, so obviously the "name of the string" is also going to store base address of the string. (base address of the character array)

→ so here 'name' is an internal pointer to the string.

→ Index will be from "0 to size-1".

VVVV Imp) In this array what we have declared char name[10] ⇒ remember we can store only 9 characters because string will always end with a null character. So at the last we always store "\0"

→ here we can store only 9 characters.

Initializing a string:

Note: generally how we write a string ⇒ "welcome" → in double quotes.

"c programming teehres" → this is also a string.

VVImp "12345" → this is also a string →

It is not like that, if a string is an array of characters you can only store a b c d e --- z, you can also store a white space, numbers, if you write 12345 in double quotes it is considered as string.

→ if you write '1' → it is considered as character.
→ If you write '2' → it is considered as character.
→ If you write ' ' → it is considered as character. (space)

→ shouldn't confuse that characters are only a b c d -- z

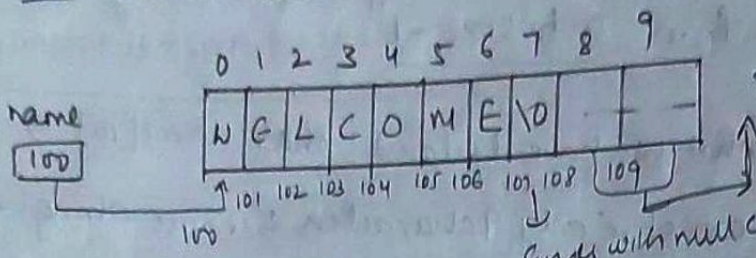
VVImp → If you write "12345" → "abcde--" → their ascii values are stored.

One way of Initializing a string:

```
char name[10] = "Welcome";
```

↓ It is called as string literal

How Welcome will be stored in the memory-?



Second way of Initializing string:

```
char name[10] = { 'W', 'E', 'L', 'C', 'O', 'M', 'E', '\0' };
```

↓ Explicitly you have to declare null value here.

and how the characters are stored above the same way here also it is stored.

Third way of Initializing a string:

```
char name[] = { 'h', 'e', 'l', 'l', 'o', '\0' };
```

↓ here we didn't mention size but we have initialized so this is also correct way of initialization.

→ Compiler will automatically find the size and according to that it will store.

In this case the size will be 6.

h	e	l	l	o	\0

6 bytes will be allocated, no wastage of space.

you can also give like:

```
char name[] = "hello";
```

here automatically the compiler will add null (\0) at the last.

size = so no of characters + one null value → this we need to take care.

→ If we write "hello world" → this is also a string, space, symbols are also allowed in a string.

`char stringname[size];` → this is similar to one-dimensional array.

you can also say:

String is a one-dimensional array or 1d array.

When each element is a character constant or a string constant. but here the datatype must be a "char".

vimp

In C string is not considered as a datatype like int, float, char-- arrays are also derived datatypes in C, but we can represent "string as a character array in C"

→ format specifier for string is %s.

Note:

Derived datatypes are derived from primitive or fundamental datatypes. (are generally used by user itself).

Another way of Initializing a string:

a). → char name[] = { 'W', 'E', ' ', 'A', 'R', 'E', 'O', 'K' }

here we are giving space here

Here the size of the string will be 8 + one null character.
= 9.

✗ → If you declare like char s[5] = { 'W', 'E', ' ', 'A', 'O', 'C' }

↓

This is not allowed.

Because size is only 5 and we are providing more values

b) char name[10];
name[10] = "Hello"; } ✗

char name[10] = "Hello";
char s1[10];
s1 = name;

} not allowed name is also string s1 is also string I want to copy hello in the name to s1 this is also not allowed.

s1 is like an array name, we cannot put array name to left side of equal to operator.

How to read a string / Initialize a string at runtime:

We can use scanf(), gets() also.

We have drawbacks also.

Valid / Invalid → Assignment to students

```
char s1[50] = "HNO-10-4-163";
char s2[] = "12345678910";
char s3[5] = "Hello";
char s4[] = {'H','e','l','l','o','\0'};
char s5[15] = "Hello World";
```

→ you can initialize / read a string at runtime using standard functions.

We have two functions to read a string: scanf, gets.

getchar is also there but we have to put that in a loop

→ we will discuss scanf and gets to read a string at runtime and we will discuss the drawbacks of both the functions and advantages too.

How to read a string using scanf function:

→ string is nothing but it is a character array which ends with null character.

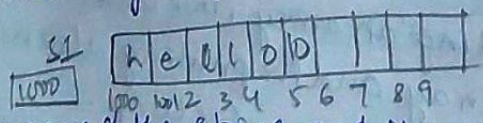
difference between character array and string:

if i declare like below:

char s1[10] = "hello"; → This is how we initialize a string at compile time using a string literal.

→ whatever you are writing in double quotes, that is considered as a string.

→ automatically compiler will do what? 10 bytes will be allocated to s1 string.

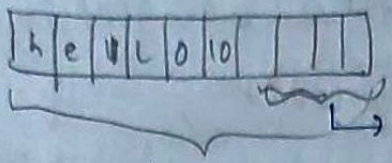


Suppose memory is from 1000 to 1009 and s1 is the name of the string and it will store Base address.

Or we can say S1 stores the 1st byte address of the string / character array.

→ and here iam going to store hello (this is what a string)

→ Compiler will automatically append a null character after a string. ('\0' means



↳ here we have some garbage value

this complete is a character array.

→ but in this character array string is what only "hello".

→ once the compiler will find this null character means it is the "End of string"

→ so this is what is the difference between character array and a string.

→ string would always end with a null character (\0).

→ otherwise string is just a character array.

String Input & output functions:

Reading string using scanf() function:

```
void main()
{
char s1[10];
printf("Enter string");
scanf("%s", s1);
}
```

↓ format specifier for reading a string is %s

↳ simply here you have to pass the name of the string s1

Difference of reading an array and a string:

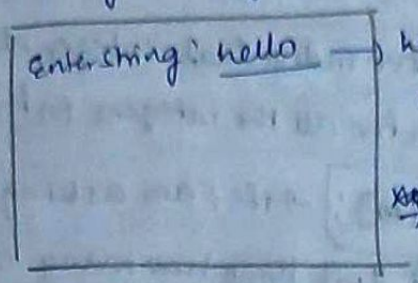
String is also an array, but it is an character array.

But when we were reading an array, we used for loop for reading an array and for printing also we used for loop.

→ But to read a string we are not using any for loop although it is an array (character array).

Suppose we are entering a string like this.

op:

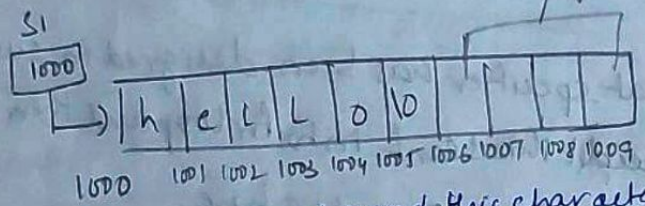


→ here we have 5 characters, so we have to print 5 characters. but we are not using any loop

→ another difference is we are not using any address of operator here.

→ Why we are not using address of (&) operator here →

Because s1 is what →



→ s1 → is storing the Base address of this character array.

→ we have stored a string hello here and after string hello we have a \0 (null character which shows that here it is the ending of a string.

→ When you pass s1 → means here you are passing 1000, which is the address of 1st one / 1st byte / 1st location.

→ so 'h' will be stored here (at 1st location), then 'e' then 'l' then 'l' then 'o'. and at last after you press Enter at run time Execution then it will store "\0" → null character at the end.

VIMP → That is why we are not passing any address of (&) operator here, because obviously we are passing address only, and we are not using any for loop → why so is because, here we don't need to specify the address for every character you are entering.

→ Suppose we take an array int a[5]; → means we are entering 5 elements of an array, and at runtime we have to enter 5 elements / 5 integers in array.

→ We were passing address for individual integer elements in the array i.e memory location for all the integers in for loop

→ But in a string `char s1[10];` → if i am asking to enter a string / enter a name, i don't know how many characters are there in a name,

→ It is not like that i specify enter 10 characters of your name, we don't know how many characters we are going to enter.

%s → This format specifier has been designed in such a way that we don't need to put this scanf in to for loop and then we don't have to put any address of (&) operator here.

How to print a string?

```
char name[10];
printf("Enter name");
scanf("%s", name);
// printing a string we use %s → it is used to read & print a string.
printf("%s", name);
```

Drawback of using scanf() to read a string!

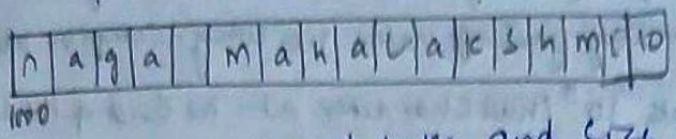
If i want to enter the name like below.

Naga Mahalakshmi

↳ here i give space suppose.

ie Naga space mahalakshmi i want to print.

→ so how it will be stored in array.



→ Suppose size we have to take and size in strings is what →

Number of characters + One for Null.

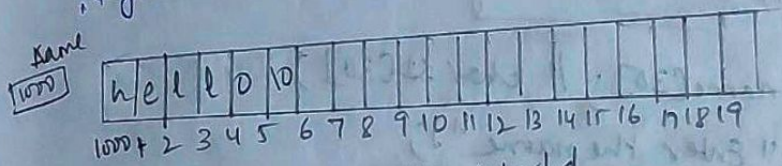
→ So we have 15 characters + 1 (we need for null).
(includes space too as space is also a character.

So size can be char name[17]; or more than that we need to declare like char name[20];

→ But output you should get Naga Mahalakshmi.
but here we get Naga only if you use "scanf" function to read this kind of string.

scanf will not consider white space in string.

So when you enter naga mahalakshmi how it will be stored in string if you use scanf function.



Enter name: hello world

If you enter the string hello and then space world here then it will take h, e, l, l, o but when you enter space, scanf would not consider space in a string so it will think hello is the end of the string and compiler will automatically put \0 character, it is not going to store the space and world here. it will take garbage values after that and when you print it will print only hello, (it will not print any garbage value). because obviously here name is what) a character array of 20 characters & we want the exact o/p obviously name.

If you are printing any string here we want exactly the exact value, not any garbage value. i.e. why compiler will automatically puts "\0" (null character at the end of the string

→ So when ever you reading i.e. when ever you are printing it is the format specifier, it is programmed like whenever it reads hello and when it reads end of the (there) string, it is not going to print which is here and this null character is one draw back of using scanf here.

To overcome this draw back we use gets function.

gets → it will read the complete string including whitespace until you press a new line or you press an Enter, it then it will read the complete line.

So if you want to read the complete line then use gets function

Using gets() function to read the string:

```
char name[30]; // char s1[30];
printf("Enter the name");
gets(name);
  s1
```

name of the string only

No format specifier, nothing else is required.

```
// printing
    printf("%s", name);
```

we can also use puts() function to print a string.

so here if you enter the string like:

Naga_Maha_Lakshmi ↵ Enter

gets → then it will read all the characters including space and whenever you press Enter, it will stop reading automatically and at the last it will put "\0" (compiler puts \0 at end)

drawbacks of both scanf and gets functions!

→ gets is better than scanf function, but both functions have a drawback called "buffer overflow" drawback.

Buffer overflow drawback:

What is this drawback →)

suppose i am writing like this:

char s1[5]; → here i am declaring a string of size 5 only suppose.

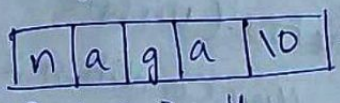
→ it means we can read only 4 characters and one is for null character.

→ And i am entering the string at run time as: naga maha lakshmi only suppose.

vv imp → and if you use gets(s1); → it will try to store the entire string although the space is allocated only 5.

ie: if: naga maha lakshmi ↵

vv imp



→ according to the logic of string it should store like this

but when you use gets(s1) → it will store the Entire string i.e. beyond the capacity and it even prints the Entire string.

So this is what is overflow, it is not going to check the buffersize.

→ memory is allocated 5 bytes, and gets is going to store beyond the memory size.

→ So whatever values are there in that memory it is going to overwrite that thing (this is very risky)

→ some very important / some critical information may be there in that memory (beyond memory which gets function is try to overwrite that one).

→ It is very unsafe and it is risky to use gets because of buffer overflow condition.

Rather than gets() it is better to use fgets().

** → same is with scanf function.

```
char name[5];
printf("Enter name");
scanf("%s", name);
printf("%s", name);
```

⇒ This also (scanf) will also do the same thing

Suppose i take "Welcome" → here it should take 7 characters and 1 would be null. means (8).

→ But here also we have declared size as 5, but here scanf also shows complete string name welcome and it will print welcome too. (unsafe & risky as gets).

→ It is also buffer overflow condition & scanf function also doesn't check the buffersize.

Note!

In scanf if you write like below:

```
scanf ("%.4s", name);
```

this will read only 4 characters → i.e. if you Enter hello it will read hel and then lo it will take.

This is one alternative, but not good practice.

printf and puts functions!

We will discuss about both printf and puts functions

These two functions are used to print a string on output screen.

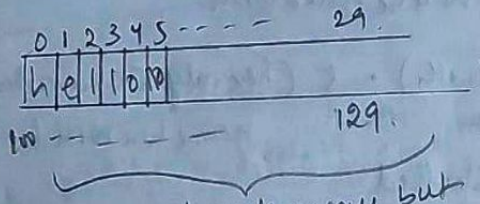
printf() function!

```
char name[30]; scanf("%s", name);
```

simple syntax of printf() function is what?

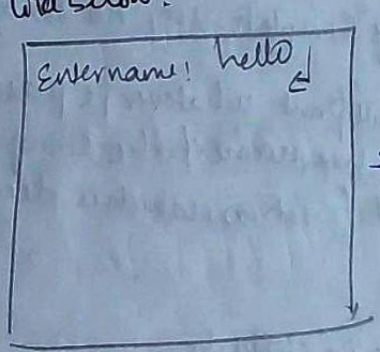
```
printf("%s", name);
```

↓ this is name of the string acts as the pointer to the string.



→ memory allocation has been allocated to this string like upto 30 bytes index would be 0 1 2 - - - 29

Suppose address is 100 then 101 - - - - 129 and suppose I have entered something like below:



→ It is asking when your going to enter a name (here string name)

→ by this scanf we can write that Enter name: using printf statement.

printf("%s", name); hello.

after you type hello lo will be added to say its the end of the string.

printf("%s", name);

printf(name) ⇒ means

In name the name of character array is name & it contains address 100, so it is pointing to the first character of the string so hello will be printed as the string.

What can be done using printf function:

you want to print specified characters, suppose i want to print only 2 characters among hello.

so if you write something like below

printf("%.2s", name); if you write 0.5s → it will print 5 characters among the complete string.

0.2s → so it will print he

Suppose if i write fieldwidth.

printf("%10.5s", name);

nagamahalakshmi

just to understand that we have fieldwidth we gave underscore. this 10 will be filled with

_ _ _ _ _ n a g a m (fieldwidth) . 5 (here only 5 characters of this name here will be printed).

you will not get underscore. this only we get and this will be right shifted.

How to use puts:

If you write here puts it is very simple to use, no format specifier only write function puts(name); // it will print whatever is passed here name of the string.

this is a pre-defined function which is declared in stdio.h header file.

meaning of puts → declared in stdio.h.

difference between printf and puts function:

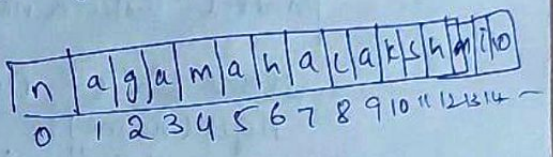
→ only difference is what ever is in puts function, it will automatically add a newline at the end of the string.

→ In printf, if you don't give newline character it prints in the same line.

→ if you put 2 puts in the same program:
it will print nagamahalakshmi and then nagamahalakshmi in next line.

→ but if you print the same string using printf 2 times it will print in same line.

→ but after printf without \n, you give puts function it prints the strings in the same line, but if you again give puts then it will print in next line.



view.
→ If you write like below:

```
printf("%s", &name);
```

↳ this will also print the string not the address.

```
printf("%s", &name[2]);
```

↳ &name[2] → means name[2] i.e it will start printing the characters from index 2 and it prints till it encounters '\0' null value.

```
printf("%s", name[2]);
```

↳ you will not get any o/p here this will be error

// program to read a string and print a string.

```
#include <stdio.h>
int main()
{
    char name[30] = "Hello"; // hello world.
    printf("%s", name);      // {'h','e','l','l','o','\0'};
}
```

Olp: Hello

// Initializing at runtime.

```
void main()
{   char name[30];
    printf("Enter a name");
    scanf("%s", name); // we need not specify & because
    //gets(name);        // name itself is a pointer.
    printf("%s", name);
    //puts(name);
}
```

Olp: Enter a name
hello
hello.

Enter a name: hello world. // scanf won't take spaces.
hello // prints only hello

buffer overflow condition also will be there scanf & gets.

// try all the puts gets, &name[2], &name all and check.

String manipulation functions: C supports a string handling library which provides useful functions that can be used for string manipulation all these functions are defined in string.h header file. Some of them

strcat(), strlen(), strcpy(), strcmp(), strlwr(), strupr(), strstr(), strrev(), strncat(),

strlen():
C program to find length of string:

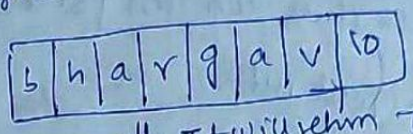
→ we will find the length of string using a predefined function strlen().

→ The meaning of strlen() is already defined → in string.h header file.

→ If you want to use any predefined functions regarding strings we have to include <stdio.h> header file.

→ We can find out the length of string without using predefined function also. (by our own logic)

suppose I take a string

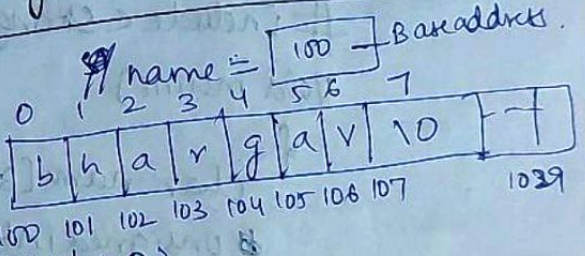


⇓ It will return 7.

→ length of string is what here: 7 characters we have (length means we are not including null character).

It is not part of string, it indicates end of string. (\0).

→ **strlen finds the length of the string excluding null character.**



Program

```
#include<stdio.h>
#include<string.h>
int main()
{ char name[30]; int count=0;
printf("Enter name");
gets(name); count=strlen(name); printf("String length = %d", count); }
```

// We have to print the length of the string so we will use strlen function
// strlen uses what → strlen (parameter is passed (it accepts string or we can say it accepts pointer to the string)

strlen function → Is a predefined function and it is written in such a way that the name of the string or the parameter we pass to the strlen function is a pointer (char name → pointer)

strlen (char* str) ⇒ it will return an integer value.
↳ pointer to a string.

→ here name is a pointer.

→ It will return an integer value or more precisely we can say it will return an unsigned integer value.

→ Why unsigned?)

Length of string cannot be a negative number.

→ the datatype of this strlen function is unsigned int.

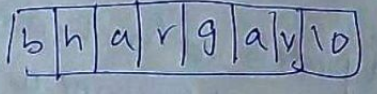
→ somewhere it is written size_t ⇒ this is nothing but unsigned int.

→ we need not bother about how strlen function code is written.

Program:

```
#include<stdio.h>
#include<string.h>
Void main()
{
  char name[30];//int count=0; we can write this also
  unsigned int count=0; //strlen returns unsigned int value
  Printf("Enter name");
  gets(name);
  count = strlen(name);
  printf("String length is: %d", count);
}
```

printing length.



Op: 7

Op: 17 (space is counted)

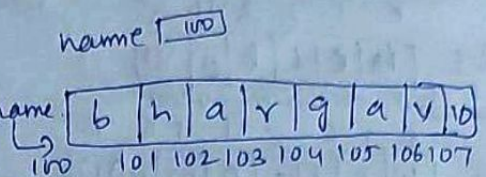
calculating length of string without using strlen() function!

program to count No of characters in the given string till we encounter \0

Logic: once we find character it will increment count by 1 until \0 is found.
character by character we have to compare till null.
so obviously we have to use loops (any loop).

Program:

```
#include<stdio.h>
#include<string.h>
int main()
{
unsigned int count=0;
  char name[30]; int i;
  printf("Enter name");
    gets(name);
  while(name[i]!='\0')
   {
     count++;
       i++;
   }
  puts(name);
  printf("%d", count); // printf("length of string is:%d", count);
}
```



Output: 7

Explanation: only while loop explanation. i=0 here.
while(name[i]!='\0') ⇒ while(name[0]!='\0') in while
name[0]=b and it is not \0 so enters loop and count=0+1=1.
and i=1 and while a[1]!='\0' ⇒ h≠'\0' so enters the loop and process continue till the loop encounters the null character and then exits the loop and then prints the complete string and count i.e the no of characters in this string.

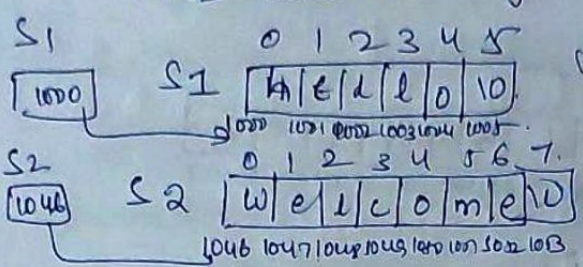
Src

// C program to Concatenate two strings.

We will use predefined function to concatenate the two strings and without using predefined functions

Concatenation means!

Let us take two strings



We have taken two strings one is "hello" and other one is "welcome"

We will declare S1 and S2 and size is what?

```
char S1[6] = "hello";
char S2[8] = "welcome";
```

In S1 we have the first byte or the base address of S1

In S2 we have the base address of S2

What do Concatenation mean?

We are going to append the S2 to S1, i want hello welcome.

This is what is concatenate or Combine two strings. we are going to append welcome.

More precisely we can say we are going to append a copy of welcome

obviously we are not going to remove w from S2 and place it in S1 and so on--

→ we are going to append a copy of welcome at the end of hello in S1

(or)

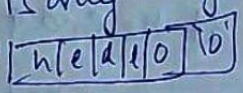
→ We can also do something like below!

like "welcome hello" also we can append this way also is ok.

But ultimately it is appending of string S1 to S2 or S2 to S1.

First point we need to take care about concatenation is what-?

suppose if you want to append welcome (S2) to hello (S1), obviously S1 should be large enough, but we don't have size in S1 and size is only 6 bytes, now when you will store this string-? (welcome).



So obviously

The size of S1 / destination string (and S2 is source) should be large enough to accomodate the string. (S2 string / source string).

So better to take size of string more.

How to use string concatenation function.

The pre-defined function is "strcat".

→ strcat function accepts two arguments, (string1, string2) or we can say (destination string, source string).

a)

If you want to append S2 string at the end of S1 string then at first we will write S1 and then S2 like below.

```
strcat(S1, S2);   // it will return pointer to destination string.
```

↑ means here the first argument is always destination string and the second will always be the source string.

and it will return what ?)

a pointer to a destination string. and finally the destination string is "hello welcome"

so strcat(S1,S2); will return pointer to the destination string and we can directly print here "S1" or you can say the destination string.

b) But if you want to append S1 string at the end of S2 string then at first we will write S2 and then S1 like below.

```
strcat(S2, S1);   // it will return pointer to destination string
```

→ source string.

↑ here this is destination string

c) strcat is a predefined function and the definitions of strcat is defined in # "string.h" header file.

→ Obviously the compiler cannot understand "strcat" meaning

→ We have to tell compiler the meaning of strcat and the meaning is already there in "string.h" header file i.e. why we use string.h header file. So that

the compiler can refer string.h and know what exactly is the meaning of "strcat"

Using predefined function (strcat) to concatenate two strings:

→ If you use pre-defined function, you have to do nothing its very simple.

But one problem with "strcat" function is what?

Buffer overflow → It is not going to check the size of destination string.

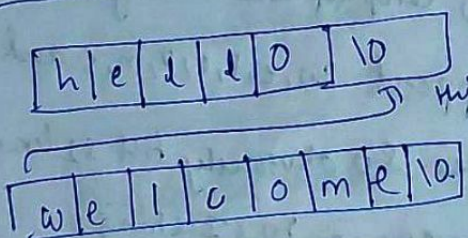
Let us explain it with example:

```
strcat(s1,s2);
```

destination string is s1 here, and suppose its buffer size is only 10:

```
char s1[10] = "hello";
```

Obviously we cannot append "welcome" to hello, because we don't have allocated size in s1, we don't have more size. but strcat will still append / concatenate this welcome at the end of hello. like below.



There is no space but still it will do, so obviously it is not safe.

program:

```
#include<stdio.h>
#include<string.h>
void main()
{
 char s1[30]="hello";
 char s2[10]="welcome";
 strcat(s1,s2);
 printf("%s", s1); // we are updating a copy of s2 to s1 so s1 is destination string, so we print s1.
}
```

h	e	l	l	o	w	e	l	c	o	m	e	\0

o/p: hello welcome.

Concatenating two strings without using string concatenation function.

Logic:

→ Obviously we have to append the first character of (s2) welcome i.e. (w) to null character (\0) of s1 string.

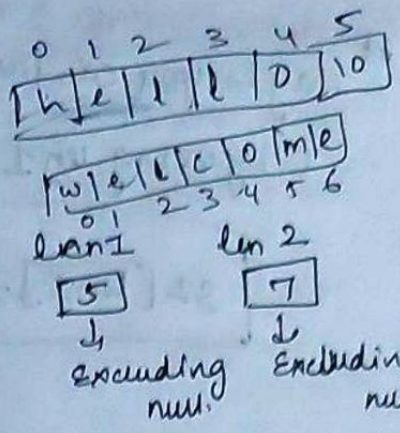
→ So we should know what is the index of '\0' in s1 string. i.e. 6 (or) we should know the length of the string (s1).

→ as well as we should know length of s2 string too, why?

because we are going to append w, e, l, c, o, m, e and then '\0' null character. So obviously till s2 length we are going to append the string to s1 string.

→ So we need to know length of both strings.

```
void main()
{ int len1, len2;
  char s1[30] = "hello";
  char s2[10] = "welcome";
  len1 = strlen(s1);
  len2 = strlen(s2);
```




```
for(i=0; i<=len2; i++)   // We are going to append one by one so we use for loop.
{                        // Upperbound condition is what?
```

/* i.e. loop should run till what?
// We are going to append s2 to s1, so we are going to
// traverse s2 till 10 (null) * What is the length of s2
// it is 7, so upperbound should be 7, till this null
// character we are going to append this. and we have to
write <= len2 because we also want null to be
appended. */

/* Now whatever is there in s2 we are going to append to s1
but how you write the logic ?
s2 index will be started from '0' only and (0 to 6) or (0 to 7) including null
so we simply write s2[i] at right side.

like ⇒ s2[i]; at right side.

But what you will write at s1 [ ] ⇒ ?
Where we append 'w' ⇒ at 5th index of s1 (at the
place of null character of s1. **
also length is 5 of s1 string
So what we can say / how we can take index of s1
is s1[len1] at left side.

11

may be you can just say len1, because index we will get (wrote) 5 by using len1.

```
s1[len1] = s2[i];
```

// if we write like this first appending is fine i.e w to the end of hello

// But next time we want the index 6th index and in s2[i], i will be 1st index i.e. 'e' i want to store in s1 at 6th index. (i is 1 because i gets incremented).

but len1 is again 5 ⇒ so it will append 'e' also

after hello [w], in this place (5th index) and w will be replaced with e. and this we don't want.

→ We also require to move s1 index too. 5 then 6 then 7... so on.

→ one by one we want to add characters.

→ so we cannot write simply s1[len1] at left side we have to add s1[len1+i];

```
    s1[len1+i] = s2[i];
}
⇒ printf("%s", s1);
}.
```

Complete program:

```
#Include<stdio.h>
void main()
{
  int len1, len2, i;
  char s1[30] = "hello";          // change the size of s2 to s1[6] and check?
  char s2[10] = "welcome";
    len1 = strlen(s1);
    len2 = strlen(s2);
  for(i=0; i<=len2; i++)
  {
  s1[len1+i] = s2[i];
  }
  printf("%s", s1);
}
```

Op: hellowelcome (It strlen gives null character it is going to print).

Note: Suppose I want to concatenate only Welc (only 4 characters) at the end of hello, if you want to do like this we have another function "Strncat"

"Strncat":

Strncat function takes 3 arguments:

1. Destination string (s1)
2. Source string (s2)
3. The no of characters you want to concatenate.

```
Strncat(s1, s2, 4);
```

Program for Strncat: // upto n characters we can concatenate.

```
#include <stdio.h>
#include <string.h>
int main()
{
    char s1[30] = "hello";
    char s2[] = "Welcome";
    Strncat (s1, s2, 3);
    printf ("string after concatenation : %s\n", s1);
    puts(s2);
}
```

Program to reverse a string:

We will use the pre-defined function (built in function) in C that is used to reverse a string. i.e. "strrev" and we write without using strrev() function also.

Strrev: // it is already declared in the header file "string.h".

There are multiple ways to reverse a string, we will discuss 2 to 3 ways.

→ although strrev is non-standard function, maybe the compiler will give error undefined reference to this (maybe in our compiler it will give some error).

→ It is because of the compiler, it is used basically in the older versions of C.

What is meant by reverse of a string:

Let us take the string

0	1	2	3	4	5	6	7
b	h	a	r	a	t	h	\0

length of string = 7 and having null character at last.

Reverse means: b will be at the place of last h
last h will be at the place of first b.
Second h → t and t → h
a → a then r. so on.

bharath if you do reverse | htarahb | this will be the reverse of the string.

Here can be two cases:

a) the length of the string is 7 means odd.
b) may be the length of the string is 8 means even.

How to do reverse-?

→ If you use the pre-defined function i.e. strrev, the prototype of this function is defined as below:

```
#include<string.h>
void main()
{
  char s1[20] = "bharath";
strrev(s1);   // what is the prototype of the function defined
              char *strrev(char *str); // in string.h
}
```

// strrev(char *str)
↳ it takes only 1 argument
// the string which u want to reverse
// or you can say the pointer to the string.
// so pointer to string (i.e. char *str.

[In this logic is written in the declaration / definition of this function.

we don't need details here. (as using pre-defined function).

And this string you want to reverse and what it will return the pointer to the same string, obviously after returning this will be a reversed string, so it will return pointer to the modified string.

```
#include<string.h>
void main()
{
  char s1[30] = "bharath";
  strrev(s1); // we are using predefined function strrev and just passing string as an argument
  printf("%s", s1);
}
```

output: htarahb.

Note:

all will think that if you print bharath ↑ from here.

h then t then a then r then a then h then b

then you think it is reverse of a string.

NO just you are printing the string from last, it is not reversing of string.

Actually reverse is what, swapping of these characters.

program to do reverse of a string without using predefined function.

one logic:

	0	1	2	3	4	5	6	
s1	b	h	a	r	a	t	h	10

The above is the string and we need to reverse this string.

s1[0] is going to be swapped (reversed) with s1[6]

s1[1] is going to be " with s1[5]

; soon. but s[3] will be same. no swapping " done

→ If you write down the loop just run the loop till index 2 and position 3 only.

→ Loop will run 3 times only no need to run for 'i' in the above string.

Length of the string is what? 7.

total length = 7.

and divide the length by 2 then answer will be 3.5 and ultimately answer we take is 3 (integer value only).

so the loop will run 3 times only.

→ We will run the loop till length divided by 2. not till null.

↳ Because, that is not the need, because we should run half and reversing will be done automatically.

If the length of string is 8 then also 8/2 = 4 till 4 only the loop will run. (till we have to do reversing).

```
void main(){ char s1[30] = "bharath"; char ch;
    l = strlen(s1);
for(i=0; i<l/2; i++)
```

s1[0] ⟷ s1[6]
s1[1] ⟷ s1[5]
s1[2] ⟷ s1[4].

```
{
// s1[i] = s1[l-1]
s1[l-1-i] = s1[i];
}
```
this is wrong.

it you do something like above:

s1[i] ⇒ s1[0] ⇒ b will be replaced with s1[6-1] s1[6].

⇒ now b is gone and in place of b we will have h.

s1[7-1-0] = s1[0] ⇒ in both places s1[6] = s1[0] ⇒ "h" we will have

so now b is not there in this string anymore and it is overwritten with h.

and if you do the next step, at both s[0] and s[6] we will have h only and this we don't want.

→ so we need to do a backup of the 'b' character.

We can take a backup of this character in any other character type of variable.

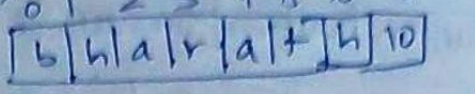
```
    ch = s1[i];
    s1[i] = s1[l-1-i];
    s1[l-1-i] = ch;
  }
  printf("%s", s1);
}
```

Complete program:-

```
#include<stdio.h> #include<string.h>
void main()
{
  char ch; int l; int i;
  char s1[30] = "bharath";
  l = strlen(s1);
  for(i=0; i<l/2; i++)
    { ch = s1[i];
      s1[i] = s1[l-1-i];
      s1[l-1-i] = ch;
    }
  printf("%s", ch);
}
```

Another logic:

If we don't understand the logic of s1[l-1-i] logic we can take i to traverse from 0 → 2 and one variable j to traverse from 6 to 4 index



i++ and j--, and once i and j both will point to same index, you have to stop.

```
#include<stdio.h>
#include<string.h>
void main()
{
int l,i,j;
char s1[30],c;
printf("Enter string");
gets(s1);
l=strlen(s1);
for(i=0;j=l-1;i<j;i++,j--)
{
c=s1[i];
s1[i]=s1[j];
s1[j]=c;
}
printf("%s",s1);
}
```

olp: Enter string: hello

olleh.

strlwr():

C program to Convert string from uppercase to lowercase.

The built-in function / pre-defined function which is used to convert uppercase letters to lowercase letters is

```
strlwr();  // string function which will convert
           uppercase to lowercase letters.
```

↳ But this is not the standard function, because on some compilers it is not going to run, because it is going to return an error, undefined reference to the above function.

```
strupr();
strstr();
strrev();
```

} these all functions we can use in older compilers.

We cannot say precisely that these are standard C functions

It is not going to run on gcc compilers.

To compile in Ubuntu:

```
gcc -g -Wall -Wextra -o filename filename.c
```

-o ⇒ specifies the name of the executable to create.

-W ⇒ options enable the common compiler warnings.

-g ⇒ includes debugging symbols.

(or)

```
gcc -o filename filename.c.
./filename.
```

Suppose the string is:

HELLO ⇒ this is in uppercase letters.

olp should be printed in lowercase i.e. hello

→ Suppose the string is like: hELLO

⇒ olp: hello.

Hello123 ⇒ olp hello123.

#HELLO123 ⇒ #hello123.

Only uppercase letters have to be converted into lowercase letters (this we need to take care while we write our own logic to convert uppercase to lowercase letter).

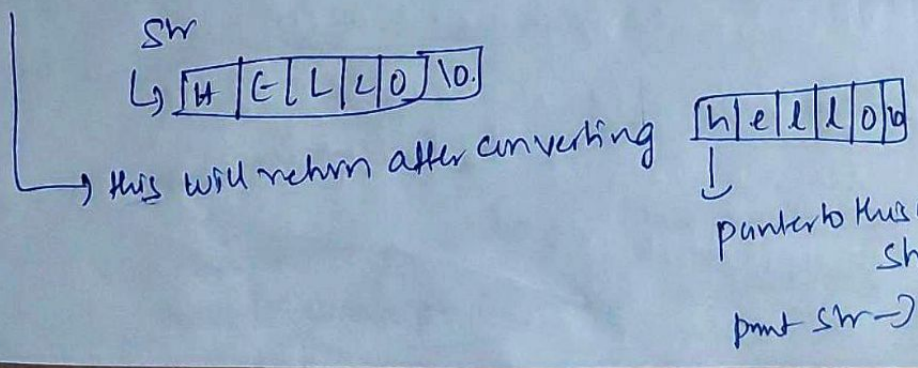
program using predefined function:

strlwr(): This function is defined in string library i.e. in string.h headerfile.

What is the prototype of strlwr():

strlwr ⇒ is going to take one argument, i.e. a character and pointer to a string which you want to convert to lowercase.

```
char* strlwr(char *str); // returns the pointer to the modified
                         string.
```



Program:

```
#include<stdio.h>
#include<string.h>
void main()
{
    char s1[30]="HELLO";
    strlwr(s1);
    printf("%s",s1);
}
```

Olp: hello

Converting lowercase to uppercase: strupr();

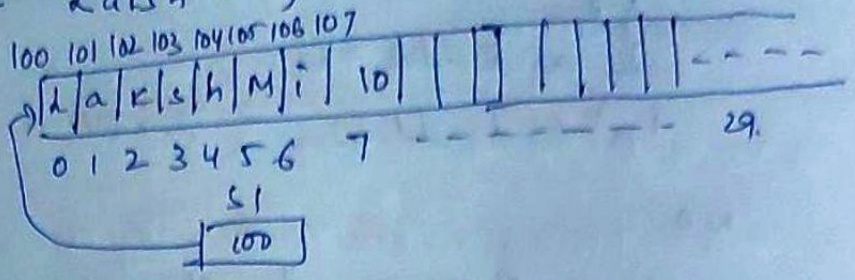
If you want to convert a lowercase string to uppercase the built-in function / pre-defined function is strupr()

```
#include<stdio.h>
#include<string.h>
void main()
{
    char s1[30]="hELlO";
    strupr(s1);
    printf("%s",s1);
}
```

Olp: HELLO

our logic to Convert a string from Uppercase to lowercase:

char S1[30] = "LakshMi";



How we will Convert -?

Obviously we will store the characters using character system.

ASCII character system in computer.

A = 65, B = 66 --- Z = 90

a = 97, b = 98 --- z = 122

Now you will convert it is not like that simply you can Check if the character is Capital letter convert into small letter - NO

We have to tell the compiler, we have to write some logic, may be some addition, multiplication, or something we have to do. some modification we have to do -?

Based on some ASCII numbers we have to do some modifications. i.e - one way.

⇒ find out the relation between Capital letter and small letter first of all then obviously we will get the idea.

A = 65 and for this if you add 32 then you will get 97 = a.

B = 66 and for this if you add 32 then you will get 98 = b

Z = 90 and for this if you add 32 then you will get 122 = z

this is the simple logic you have to apply if you have to convert Uppercase to lowercase.

but it is not like:

simply access the array element and like 'L' and add 32

'a' and add 32 ---so on.

NO

We have to check whether the character is uppercase / lowercase and if it is uppercase then only we have to add 32.

Suppose a is small case and 97 for this if you add 32, we may get some special character which we don't want.

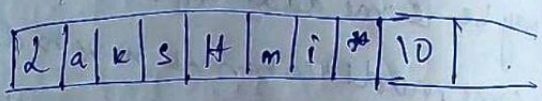
So we have to put some condition, before writing the logic.

If you write Lakshmi I should get Lakshmi I only.

i.e only uppercase letters have to be converted.

```
#include<stdio.h>
#include<string.h>
void main()
{
  int i;
char s1[30] = "LakshMi*";
        // character by character we are going to check string until NULL
for(i=0; s1[i]!='\0'; i++)   // you can put the condition in many ways.
  {
    // We have to add 32 in these characters but based on certain condition we. It should be only uppercase character then only we should add.

    How to check?
```



```
if(s1[i]>=65 && s1[i]<=90)      // should not use logical OR
                                // have to use logical AND
  {
    s[i] = s[i]+32;
  }
}
printf("%s",s);
}
```

Op: Laxmi *

c program to find substring in a string: strstr()

⇒ We have to find a substring in a given string. i.e.

> In a given string, if we want to find out a particular substring whether it is there in that string or not.

Suppose:

Given string is: ab abab bb aa bb a b abb baa.

substring: aabb.

We have to find whether "aabb" is there are not.

⇒ for this we will compare the substring's 4 characters with the main string characters.

⇒ substring length = 4 (here).

> We will compare by 4 characters in length with the main string.

So first 4 characters in main string i.e. abab are compared with aabb (substring) then are not matched

then you have to compare next 4 characters that does not mean you have to move 4 characters ahead & check

→ NO

you have to check: baba with aabb → No match
next ab ab with aabb --- so on you have to
compare like this.

→ We will take 2 indexes i and j where i represents main string index and j represents substring index

j index = 0 to 3 only here.

→ Substring length can be something else also → yes it can be

→ We have to write the logic according to the length of substring.

⇒ If suppose l=4 then 0 to 3 but if it is 5 it should be 0 to 4 so we have to take j in general as

j = 0 to l-1

⇒ But what will be the value of i → until what it will run
i value starts at 0, and ends at 10 null (last value)

it will run in a loop

```
for (i=0; str[i] != '\0'; i++)   // to traverse the string only.
    str[i+l-1];
{ // every time we run the above for loop, we take one
// variable k in which we store the value of i
    k=i;
// we take j=0 and j<=l-1; j++)
for (j=0; j<=l-1; j++)
    // p<7.0
```

```
// let us denote mainstring as 'str' and substring as 's'
    if(str[k] != s[j])
        break;
    k++;
  } if(j==l)
    return(i);
 }
 return(-1);
}
```

program to find substring is present / Not in mainstring without using predefined function.

```
int indexofsubstring(char str[], char s[])
 { int i,j,k;
 l=strlen(s);
 for(i=0;str[i+l-1];i++)
  { k=i;
  for(j=0;j<=l-1;j++)
   {
     if(str[k] != s[j])
        break;
     k++;
   }
  if(j==l)
    return(i);
  }
 return(-1);
}
```

```
void main()
{
        int index;
 index = indexofsubstring("ababababbaabbabbbaa", "aa");
 if(index==-1)
   printf("Substring not found");
 else
   printf("substring found at index=%d", index);
}
```

To find out whether given substring is present in main string using strstr() predefined function:

```
#include <stdio.h>
#include <string.h>
/*
   strstr(mainstring, substring) ---> to find substring
      in mainstring */
int main()
{
   char mainstring[40]="learn programming in c";
   char substring[30]="programming";
/* strstr() function will return the pointer to the first
   occurence of the substring, so the result has to be stored
   in pointer so we have to declare a pointer */
   char *result;
   result=strstr(mainstring, substring);
if(result) { printf("the string is present"); }
   else { printf("string is not present"); } return(0);}
```

program:

```
#include<stdio.h>
#include<string.h>
int main()
{
  char mainstring[40] = "learn programming in c";
  char substring[30] = "programming";
  char *result;
  result = strstr(mainstring, substring);
  if(result)
  {
    printf("The string is present");
    printf("\n Remaining string is %s", result);
  }
  else
  {
    printf("string is not present");
  }
  return(0);
}
```

Op:- The string is present
Remaining string is programming in c.

strstr() ⇒ is used to find out whether the given substring is present in main string or not.

P.T.O

Arrays of strings

- A string is a 1-D array of characters.
- An array of strings is a 2-D array of characters.
- Just like we can create a 2-D array of int, float etc.
- We can also create a 2-D array of character or array of strings.

Here is how we can declare a 2-D array of characters.

suppose:

```
char ch_arr[3][10] = {
                        {'s', 'p', 'i', 'k', 'e', '\0'},
                        {'t', 'o', 'm', '\0'},
                        {'j', 'e', 'r', 'r', 'y', '\0'}
                     };
```

⇒ It is important to end each 1-D array by the null character, otherwise it will be just an array of characters.
We can't use them as strings.

⇒ Declaring an array of strings ~~above~~ this way is rather tedious.
That's why C provides an alternative syntax to achieve the same thing.

The above initialization is equivalent to:

```
char ch_arr[3][10] = { "spike", "tom", "jerry" };
```

The first subscript of the array i.e. 3 denotes the number of strings in the array and the second subscript denotes the maximum length of the string.

→ In C, each character occupies 1 byte of data, so when the compiler sees the above statement it allocates 30 bytes (3 * 10) of memory.

→ We already know that the name of an array is a pointer to the 0th element of the array.

What is the type of ch_arr?

The ch_arr is a pointer to an array of 10 characters

or

int (*) [10].

Recap: ∴ If ch_arr points to address 1000 then ch_arr+1 will point to address 1010.

From this, we can conclude that:

ch_arr+0 ⇒ points to the 0th string or 0th 1-D array.

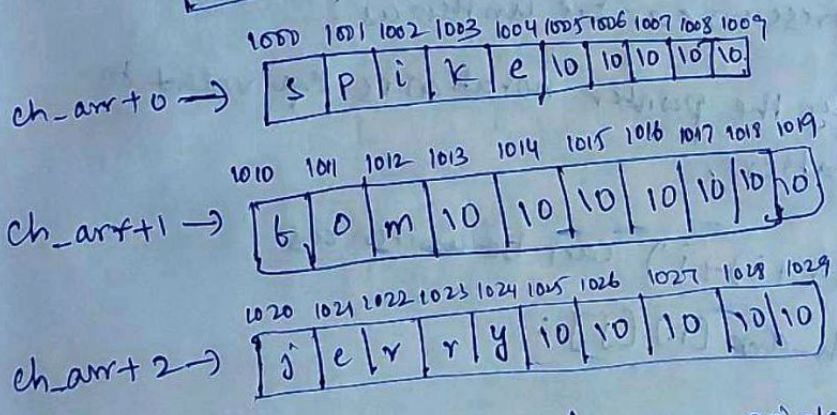
ch_arr+1 ⇒ points to the 1st string or 1st 1-D array.

ch_arr+2 ⇒ points to the 2nd string or 2nd 1-D array.

General:

In general,

ch_arr+i ⇒ points to the ith string or ith 1-D array.



We know that when we dereference a pointer to an array, we get the base address of the array.

So, on dereferencing ch_arr+i we get the base address of the ith 1-D array.

From this we can conclude that:

* (ch_arr+0) + 0 ⇒ points to 0th character of 0th 1-D array (i.e. s)

* (ch_arr+0) + 1 ⇒ points to the 1st character of 0th 1-D array (i.e.) p.

* (ch_arr+1) + 2 ⇒ points to the 2nd character of 1st 1-D array (i.e. m).

In general, we can say that:

***(ch_arr+i)+j** points to the jth character of ith 1-D array.

Note that the base type of *(ch_arr+i)+j is a pointer to char or (char*), while the base type of ch_arr+i is array of 10 characters or (int (*) [10].

⇒ To get the element at jth position of ith 1-D array.

just dereference the whole expression *(ch_arr+i)+j

***(*(ch_arr+i)+j)**

The above expression can be written as:

// In 2d-array the pointer notation is equivalent to subscript notation.

So *(*(ch_arr+i)+j) can be written as

ch_arr[i][j];

program that demonstrates how to print an array of strings:

```
#include <stdio.h>
int main()
{
  int i;
  char ch_arr[3][10] = { "spike", "tom", "jerry" };
  printf("1st way \n\n");
  for(i=0; i<3; i++)
  { printf("string = %s\t address = %u\n", ch_arr+i, ch_arr+i);
  }
  return 0;
}
```

o/p:

string = spike address = 2686736
string = tom address = 2686746
string = jerry address = 2686756

Invalid operations on an Array of strings.

```
ch_array[0] = "type";  // invalid.
ch_arr[1] = "dragon";  // invalid.
```

Here we are trying to assign a string literal (a pointer) to a constant pointer which is obviously not possible.

To assign a new string to ch_arr, use the following methods:

```
strcpy(ch_arr[0], "type");  // valid.
scanf("%s", ch_arr[0]);  // valid.
```

⇒ program:

We shall create another simple program, which asks the user to enter a username. If the username entered is one of the names in the master list then the user is allowed to calculate the factorial of a no, otherwise an error message is displayed.

```
#include <stdio.h>
#include <string.h>
int factorial(int);
int main()
{
  int i, found=0, n;
  char master_list[5][20]={"admin", "tom", "bob",
    "him", "jim"}, name[10];
  printf("Enter username:");
  gets(name);
  for(i=0; i<5; i++)
  {if(strcmp(name, master_list[i])==0)
    { found=1;
      break;
    }
  if(found==1)
  {
    printf("\n welcome %s! \n", name);
    printf("\n Enter a no to calculate the factorial:");
      scanf("%d", &n);
      printf("factorial of %d is %d", n, factorial(n));
  }
  else
    { printf("Error, you are not allowed to run this
                program");
    }
  return 0;
}
```

```
int factorial(int n)
{ if(n==0)
  { return 1;
  }
  else
  {
    return n* factorial(n-1);
  }
}
```

Op:

Enter username: admin.

Welcome admin!

Enter a no to calculate the factorial: 4

Factorial of 4 is 24.

2nd run:

Enter username: jack.

Error: You are not allowed to run this program.

How the program works-)

- The program asks the user to enter a name.
- After the name is entered it compares the entered name with the names in the master_list array using strcmp() function. If match is found then strcmp() returns 0 and the if condition strcmp(name, master_list[i])==0 condition becomes true.
- → The variable "found" is assigned a value of 1, which means that the user is allowed to access the program
- → The program asks the user to enter a number and displays the factorial of a no.
- → If the name entered is not one of the names in the master_list array then the program exits by displaying an error msg.

Array of strings in C with examples:

It is actually a two dimensional array of chartype

Ex: char names [6] [30];

In this → names is an array of strings which contains 6 string values.

→ Each of the string value can contain maximum 30 characters.

Initialization of array of strings:

- We can initialize an array of strings just as we initialize a normal array.
- The way to initialize an array of strings is

```
char var_name[R][C] = { list of string values };
```

char → specifies that we are declaring an array of strings.

var_name → refers to the array of strings defined by the programmer.

[] [] → pair of square brackets specifies that it is an array of strings.

R → Specifies the maximum no of string values which can be stored in a string array

C → Specifies the maximum no of characters which can be stored in each string value stored in string array.

list of string values → specifies various string values which are to be stored into string array.

The list of string values must be enclosed between braces.

Example:

```
char names[3][20] = {"Lakshmi", "Abhi", "Amit"};
```

In the above example, string array names have been initialized with 3 values "Lakshmi", "Abhi", "Amit".

Program to initialize an array of strings:

```
#include<stdio.h>
int main()
{
  char names[3][20] = {"Lakshmi", "Abhi", "Amit"};
  int i;
  printf("\n Elements of string array are:");
  for(i=0; i<3; i++)
    printf("\n%s", names[i]);
  return(0);
}
```

Output:

Elements of string array are.

Lakshmi
Abhi
Amit.

// Reading an displaying array of strings

Program:

```
#include<stdio.h>
int main()
{
char names[3][20];
int i;
printf("\nEnter 3 string values\n");
for(i=0;i<3;i++)
 gets(names[i]);
 printf("\n Elements of string array are");
for(i=0;i<3;i++)
printf("\n %s", names[i]);
return(0);
}
```

Op:

Enter three string values.

Lakshmi
Abhi
Amit.

Elements of string array are.

Lakshmi
Abhi
Amit.