C Language By Neso Academy

"Emon SUST 01714076452"
**** U need basic of programming to read this.*

Features:

1. Procedural lang (means divide gigantic program into small program called procedure or function) -->Main Feature.
2. Diff bet Low and high level lang simply points to "degree of abstraction" .
    1. Low level means more effort need for user, high means less effort of user. Such as if saving a file. Only clicking save. Don't need to know what is going on background(each ans every internal details about the computer).
3. C is middle level lang, Cause it allow us to access system level feature. Such as…
    1. direct access to memory through pointer.
    2. Bit manipulation by bitwise operator.
    3. Writing assembly code within C code.

For these features c used for system level app(kernel drivers, OS).

getchar():
1. used to read a single character. This is useful in simple input-handling, like menus or character-by-character input.
2. To pause the program (e.g., "press any key to continue")
3. returns an `int`, not `char`, because it might return `EOF` (end of file)

#include<stdio.h> called pre-processor & header file(.h file)(which start with "#"): Replace text with actual content.

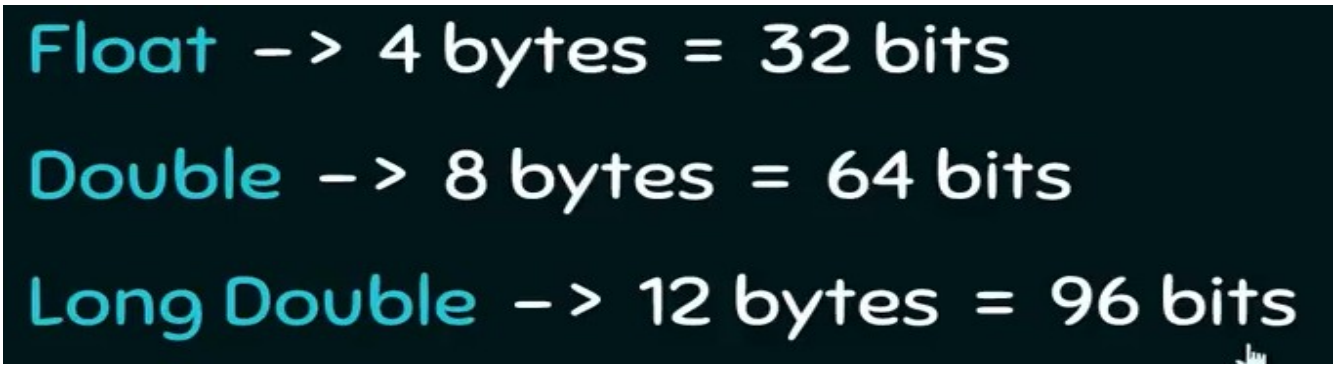
1. Replace before compilation begin.
2. Compiler convert source code intp machine code.
3. <Stdio.h> header file contains declaration(on **header file**) of function like printf, scanf.

Header file Vs C std Library: C std library contains the actual definition of the function like printf, scanf.
***then why we need diff file one for storing declaration of function and one for storing definition.?
Ans(L-2, 12min): Cause there is a program called 'Linker' maps down the declaration mentioned by the preprocessor to the actual code written in the library. It simply maps, not copy paste. This linking make fater while compiling.
If all declaration and definition store on the same file. It will replace text on actual contest what actually preprocessor done. Means definition will be add on source code. It increases the size of all code. It will be hazy. Computer has to do lot work to compile the program. Speed of compilation will be fall down. Therefore maintaining header file and standard library separately is essential.

Data type: simple means how much space a variable going to occupy in a memory. Requesting to compiler to allocate memory for the variable. (**%d** is placeholder for variable).

```c
float var1 = 3.1415926535897932;
double var2 = 3.1415926535897932;
long double var3 = 3.141592653589793213456;

printf("%d\n", sizeof(float));
printf("%d\n", sizeof(double));
printf("%d\n", sizeof(long double));
printf("%.16f\n", var1);
printf("%.16f\n", var2);
printf("%.21Lf\n", var3);
```

sizeof(int) is unary operator, not function.
Modifier: short, long, signed, unsigned.
*** INT_MIN, INT_MAX(%d), UINT_MAX(%u), SHRT_MIN(%d) need <limits.h> header file.
** char declaration will b single quote(‘’), **char a=’c’;**

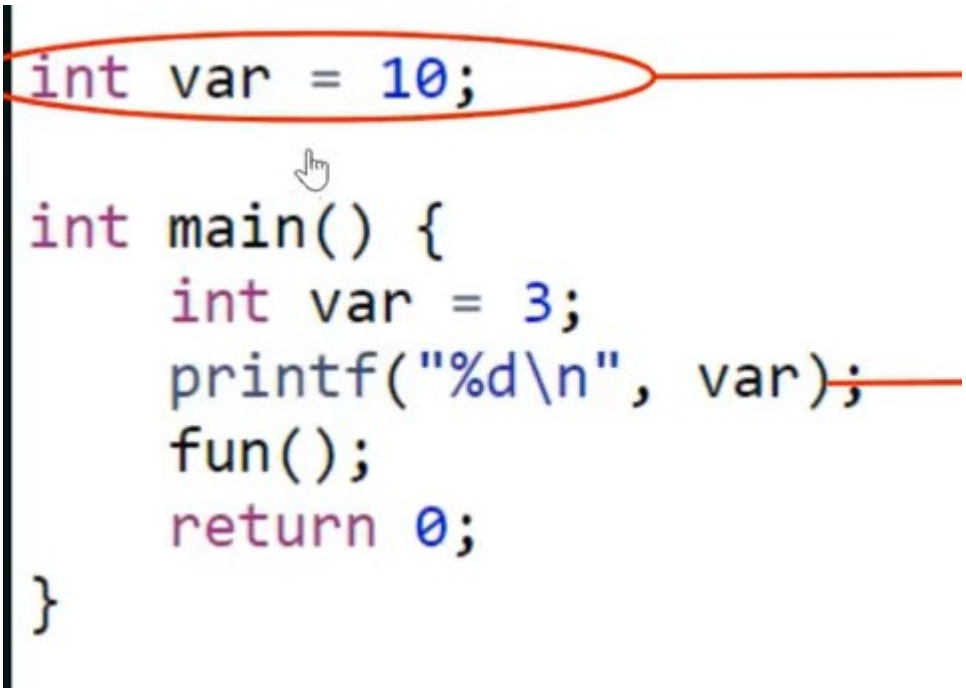


* Float precision upto 7 digits.
* Double precision upto 15 digits.
* Long Double precision upto 19 digits.

```c
int var = 4/9;
printf("%d\n", var);

float var1 = 4/9;
printf("%.2f\n", var1);

float var2 = 4.0/9.0;
printf("%.2f\n", var2);
```

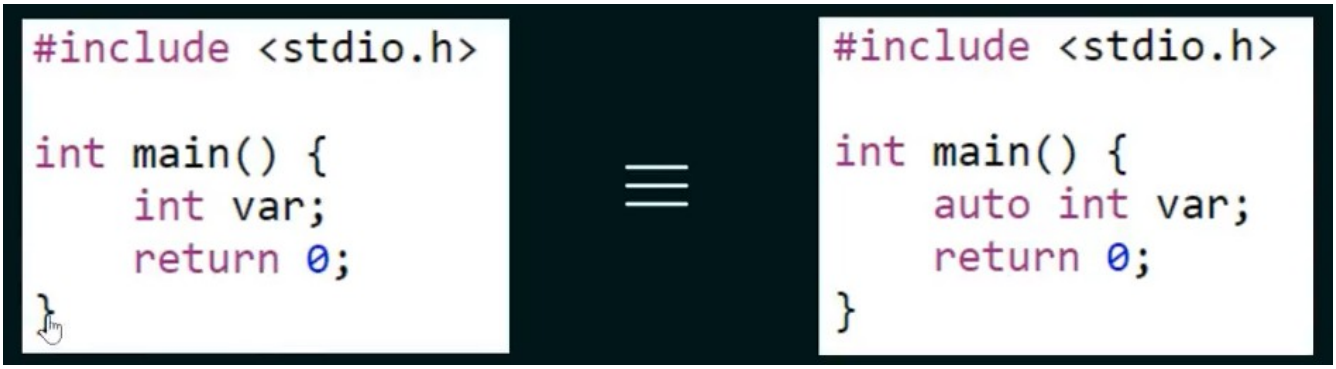
```
0
0.00
0.44

Process returne
Press any key t
```

4.0 & 9.0 will make the variable double. To make it float use **4.0f & 9.0f.**
*** Printf not only print the content on screen, but also return the number of character that it successfully printed.
** printf(“%10s”, “Hello”); will print→” Hello” // 5 space before hello.
Global variable: variable which is declared outside of all function. Global variable by default initialize to 0.
* if we declare a global variable on a file of a project(where there are many files) then this global variable will be available to all file as global variable. If such variable we don’t want to share with other file on the project then we can use static modifier. **Static int c;**



Var inside main function will get preference over global variable.
Int var; This is declaration and definition of a variable. Declaration means here are some variable of integer type, definition means asking compiler to allocate some memory for the variable.

Auto & Extern Modifier: Variable declared inside a scope by default are automatic variable. Called auto cause it automatically destroyed after completion of function/scope. So no waste of memory.



Extern modifier has only declaration, not definition. No allocation of memory. Used when a particular file needs to access that variable from another variable. This save memory and reuseability. Declaring multiple time of extern variable is allowed. Cause it is not defining.
** Multiple definition are not allowed, like int var;
int var;
int var;
When a extern variable is initialized, its memory is then allocated.
See Lec 13- 8.00 min

Register Modifier: Register keyword hint compiler to store variable in register memory. Simply write **Register** in front of a variable does not assure to save it on register memory. This is the choice of compiler whether it have to save on register memory or not. Most frequently used variable usually saved on register memory.

Static Modifier: if we declare a global variable on a file of a project(where there are many files) then this global variable will be available to all file as global variable. If such variable we don't want to share with other file on the project then we can use static modifier. **Static int c;**
See Lec-15, 14.00 min.
* When we declare a variable with static in local, it will initialize to 0, whereas without static modifier a variable declared, it will initialize with garbage value.

```
int main(){
    static int a;
    int b;
}
```

'a' will initialize to 0, and 'b' initialize to garbage value.
Static variable retain there value between function call. **See code below**

```c
#include<stdio.h>

int inc(){
    static int c;
    c++;
    printf("%d ", c);
    return c;
}
int main(){
    int v;
    v=inc();
    v=inc();
    v=inc();
    printf("%d", v);
    return 0;
}
```

ans: 3.

Below example: here ‘I” will not retain its value on every time the function called. **See lec 126(pointer program problem 8)**

```
void printab ()
{
     static int i, a = -3, b = -6;
     i = 0;
     while (i <= 4)
     {
          if ((i++)%2 == 1) continue;
          a = a + i;
          b = b + i;
     }
     swap (&a, &b);
     printf("a = %d, b = %d\n", a, b);
}
```

First function call: `v = inc();`

c Copy code

```
static int c; // c is initialized to 0 (only o
c++; // c becomes 1
printf("%d ", c); // Prints "1"
return c; // Returns 1 (stored in v)
```

Memory State after 1st call:

`c = 1` (retained)

Second function call: `v = inc();`

c Copy code

```
c++; // c was 1, now becomes 2
printf("%d ", c); // Prints "2"
return c; // Returns 2 (stored in v)
```

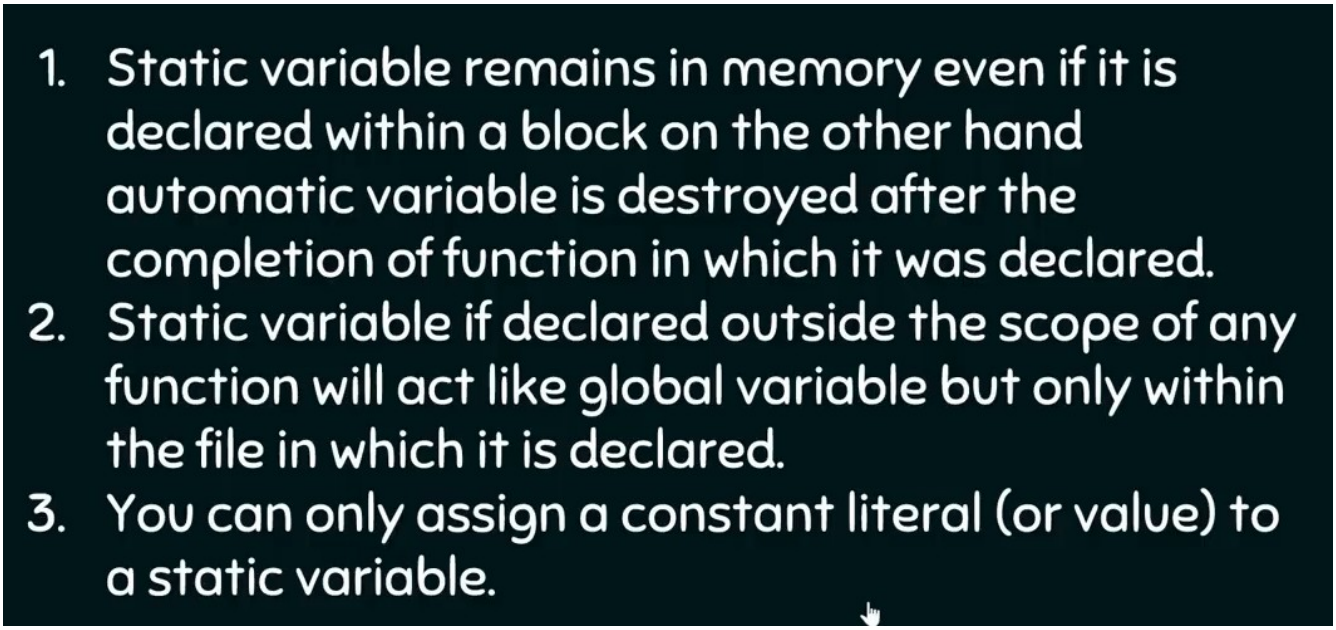
Understanding `static` Variable Behavior

1. What does `static int c;` do?

- **Static variables are initialized only once** (to `0` if not explicitly initialized).
- They retain their value **between function calls** instead of being reinitialized each time the function runs.

In C programming, a static variable is declared using static keyword and have the property of retaining their value between multiple function calls. It is initialized only once and is not destroyed when the function returns a value. It extends the lifetime of the variable till the end of the program.

https://www.geeksforgeeks.org/static-variables-in-c/



----------Static Variable Chapter Ends----------

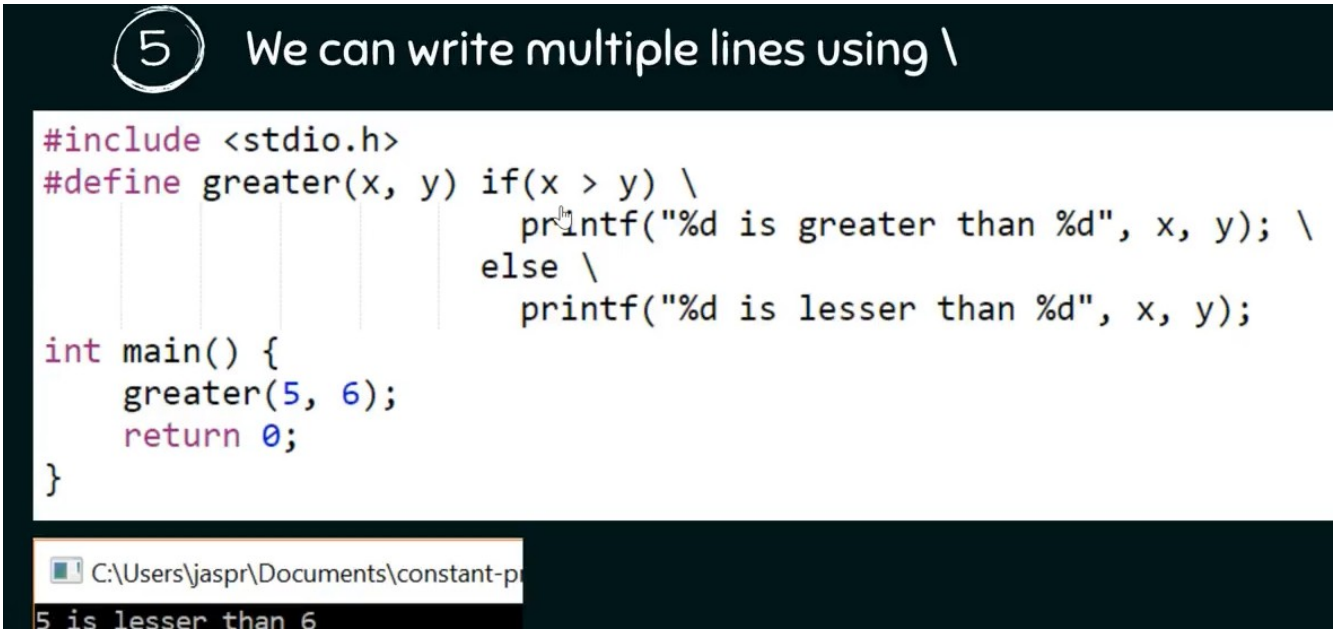
Const: Which value cannot be modified. 2 ways to use

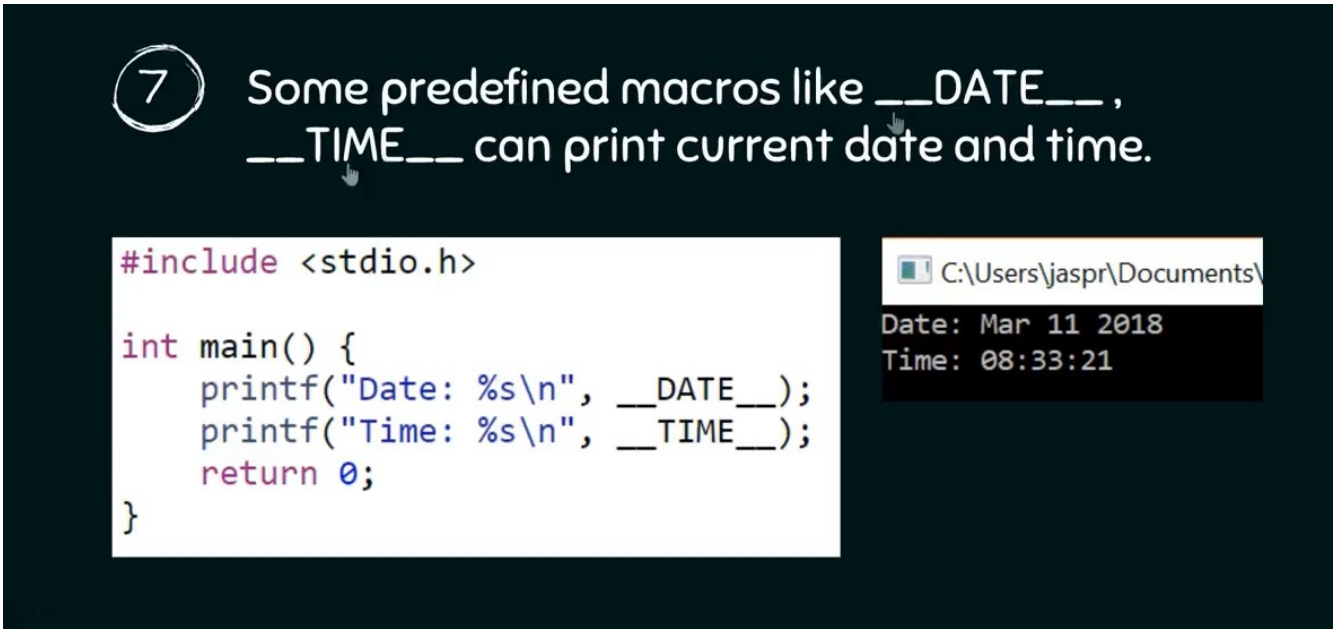
```
#define pi 3.1415 (preprocessor/macros)
const int pi=3.1415;
```

***Job of preprocessor(Not compiler) is to replace name with value.

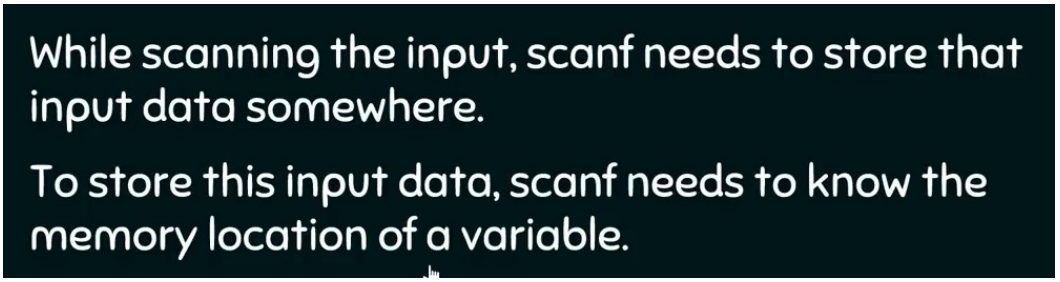
```
#define pi 1234
int main(){
        int c=3+pi;       // int c=3+1234; (name replaced with value)
}
```

----------x----------





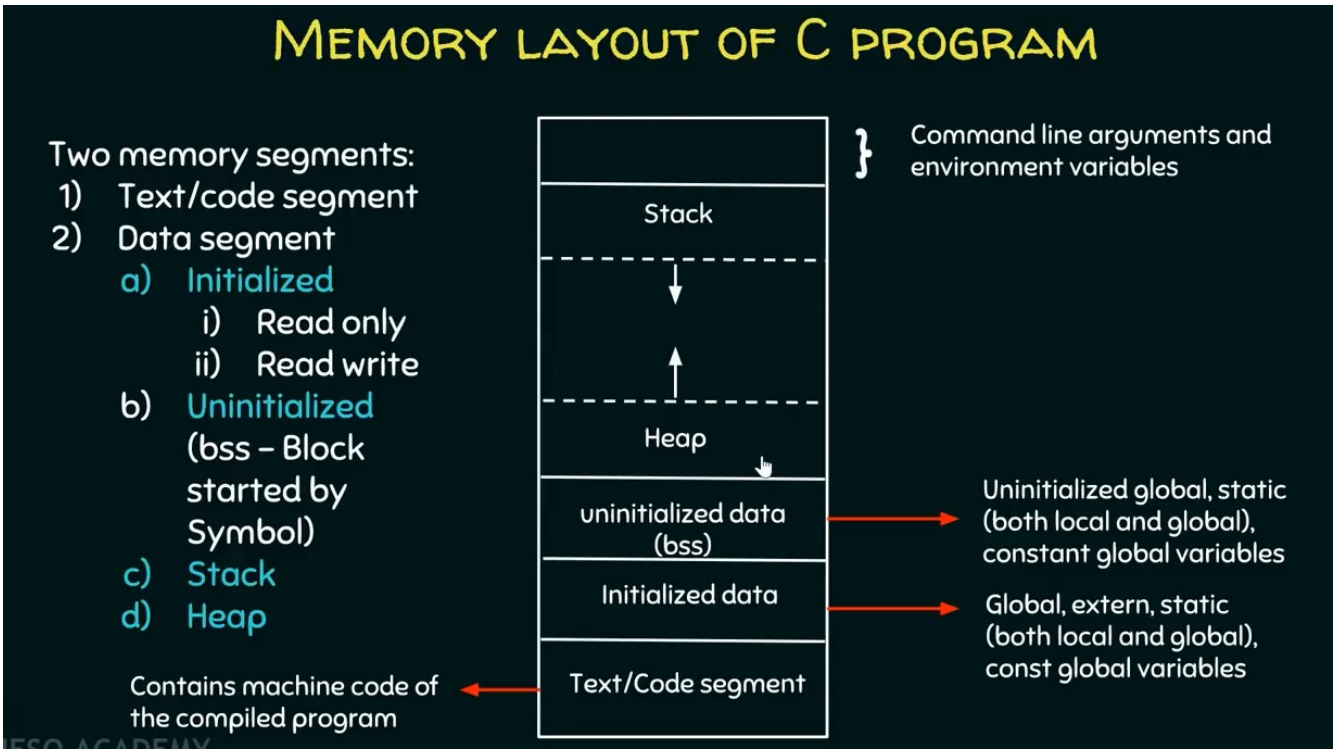
***Why Need '&' on **scanf?**



***
int a=053; // represent octal number
int a=0x43aa; // represent hexadecimal number

----------x----------

Memory Layout:

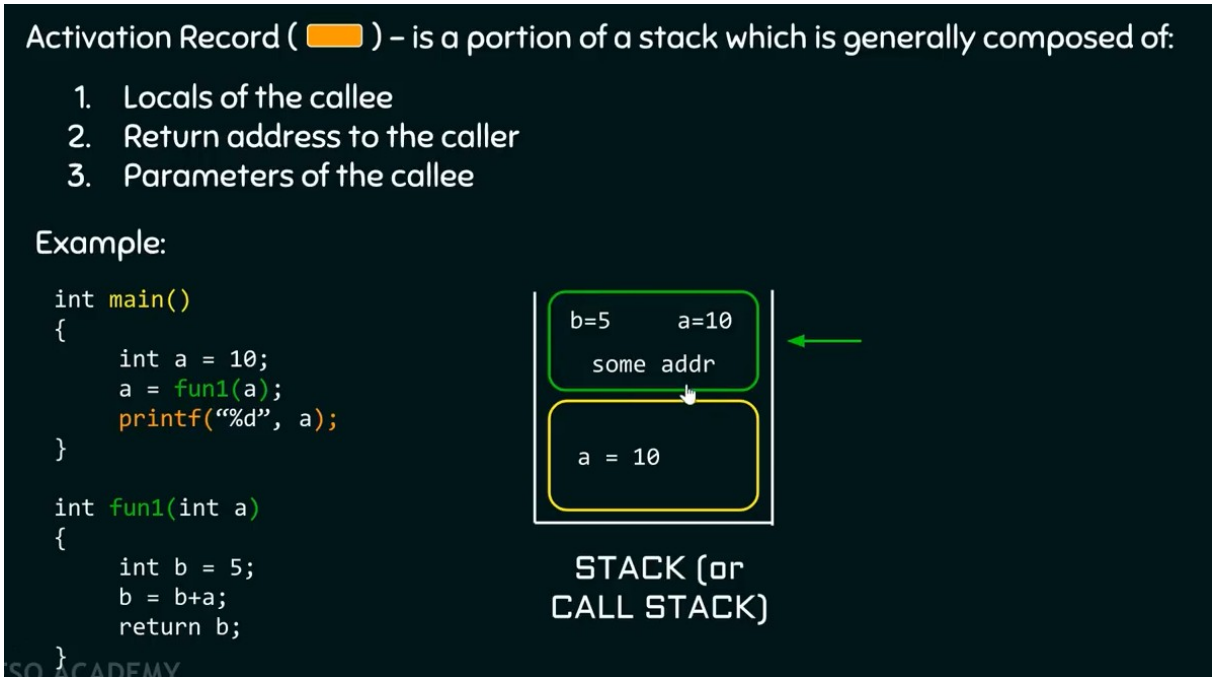


See Lec-20 for memory maping understanding.

Stack: (Lec-69(Static & Dynamic scoping))
* Stack is container/ memory segment which holds some data. Data is retrived in it in LIFO fasion.
Two Operation- **PUSH, POP**.
Whenever we call a function it will store on **Stack**. In reality it not the function which stored in stack. It is the "**Activation Record**" which maintained inside the Called stack/stack.

Activation Record:

1. Local variable of the function.
2. Return address means- where it need to go back when this function is executed. In below example when ‘fun1’ function execution completed it return back to “a=fun1(a);” of main function through return address.



```c
#include<stdio.h>

int a=34;
a=23;
int main(){
    printf("%d", a);
}
```

```
$ gcc prog.c -Wall -Wextra -std=gnu11
```

Run Share

```
prog.c:4:1: warning: data definition has no type or storage class
    4 | a=23;
      | ^
prog.c:4:1: warning: type defaults to 'int' in declaration of 'a' [-W
prog.c:4:1: error: redefinition of 'a'
prog.c:3:5: note: previous definition of 'a' with type 'int'
    3 | int a=34;
      |     ^
```

Assigning a value in global variable is not allowed. Compiler automatically set ‘int’ before ‘a’ while assigning. Thats why redefinition says. Lec-20

** Assignment is allowed on function/on a scope block only, Not globally

----------xx---------

Operator:

Name of operators	Operators
Arithmetic operators	+ , - , * , / , %
Increment/Decrement operators	++, --
Relational operators	== , ! = , < = , > = , < , >

Name of operators	Operators
Logical operators	&& , || , !
Bitwise operators	& , ^ , | , ~ , >> , <<
Assignment operators	= , += , -= , *= , /= , %= , <<= , >>= , &= , ^= , |=

-------x-------

a++, ++a:

Pre - means first increment/decrement then assign it to another variable .

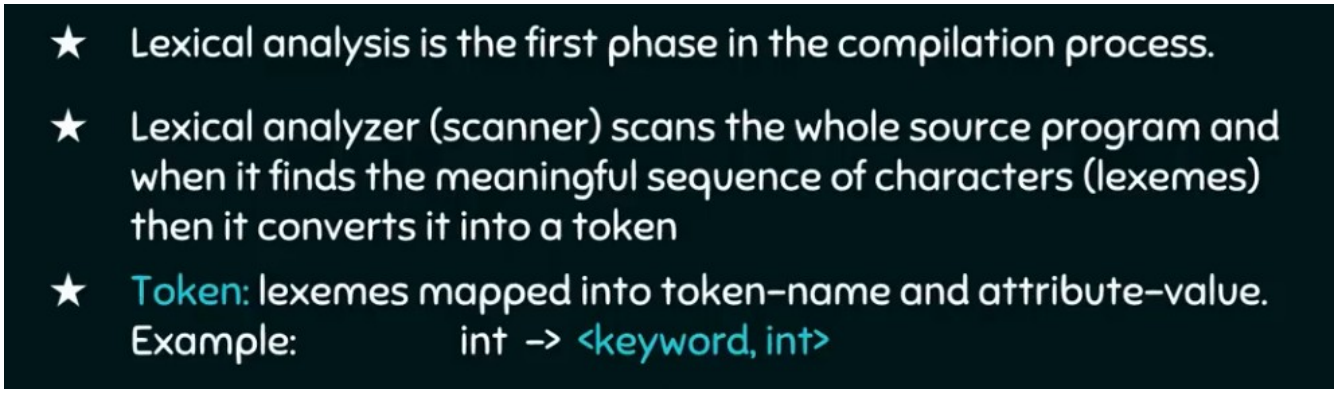
Post - means first assign it to another variable then increment/decrement.

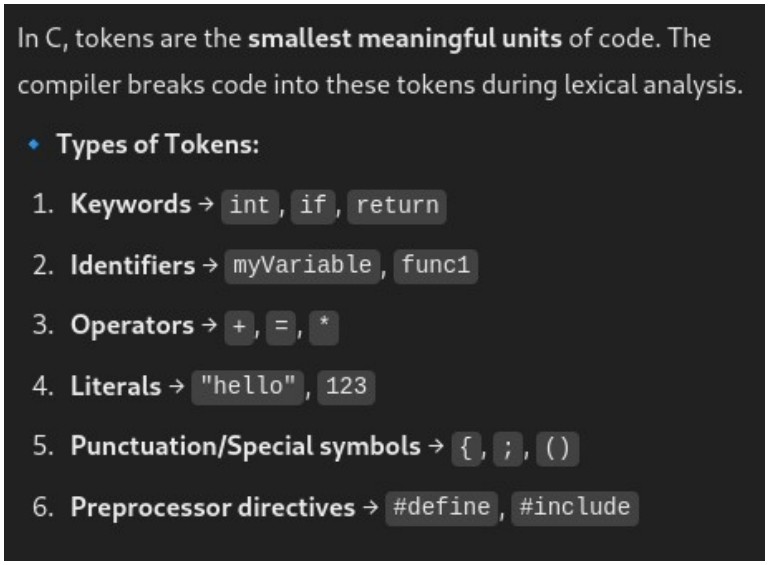
Below code output is **“5 7”**

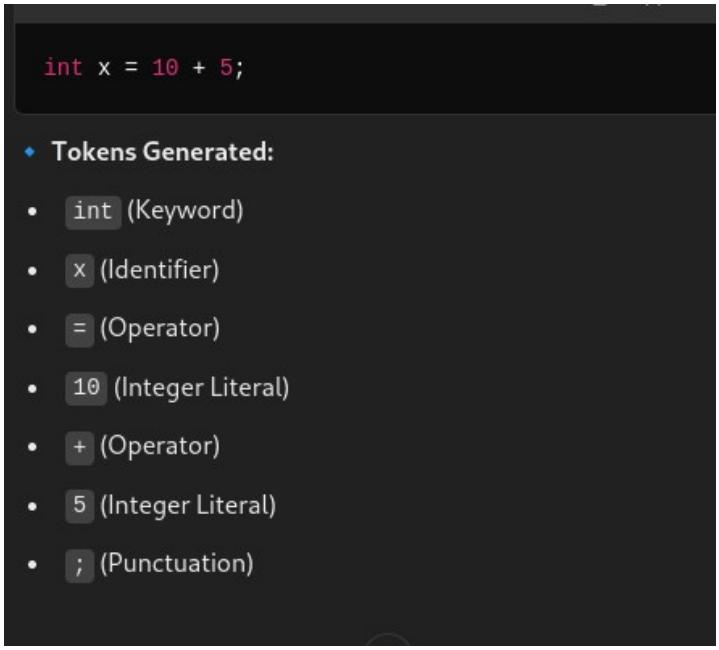
```c
#include<stdio.h>

int main(){
    int x=5;
    int xx=x++;
    int xxx=++x;
    printf("%d %d", xx,xxx);
}
```

--------x-------

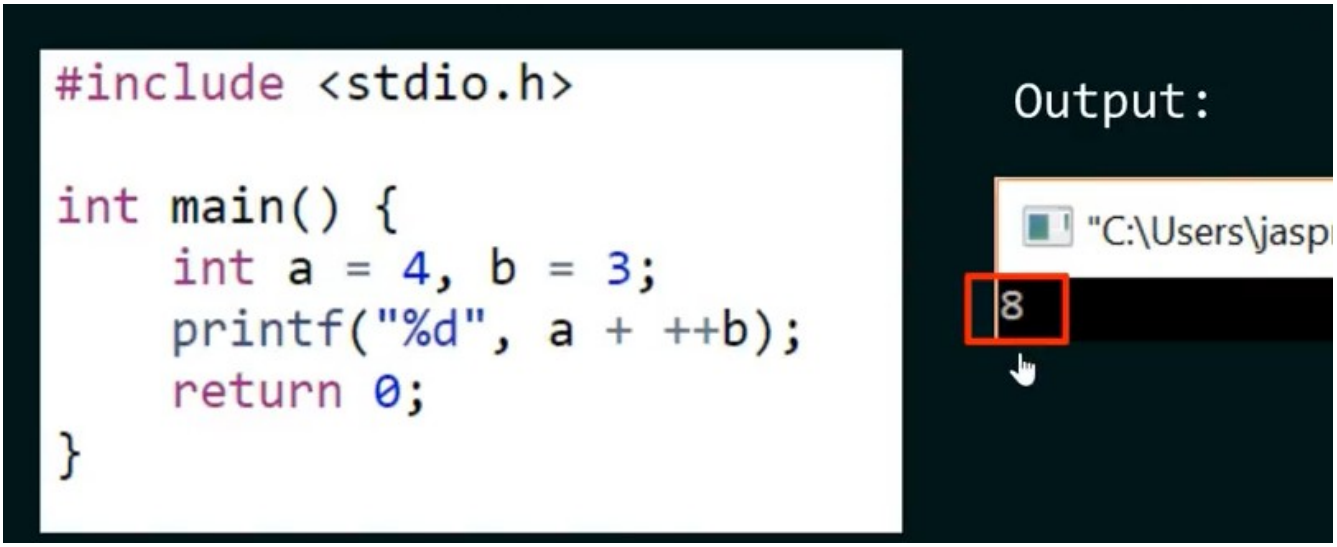
Token Generation:

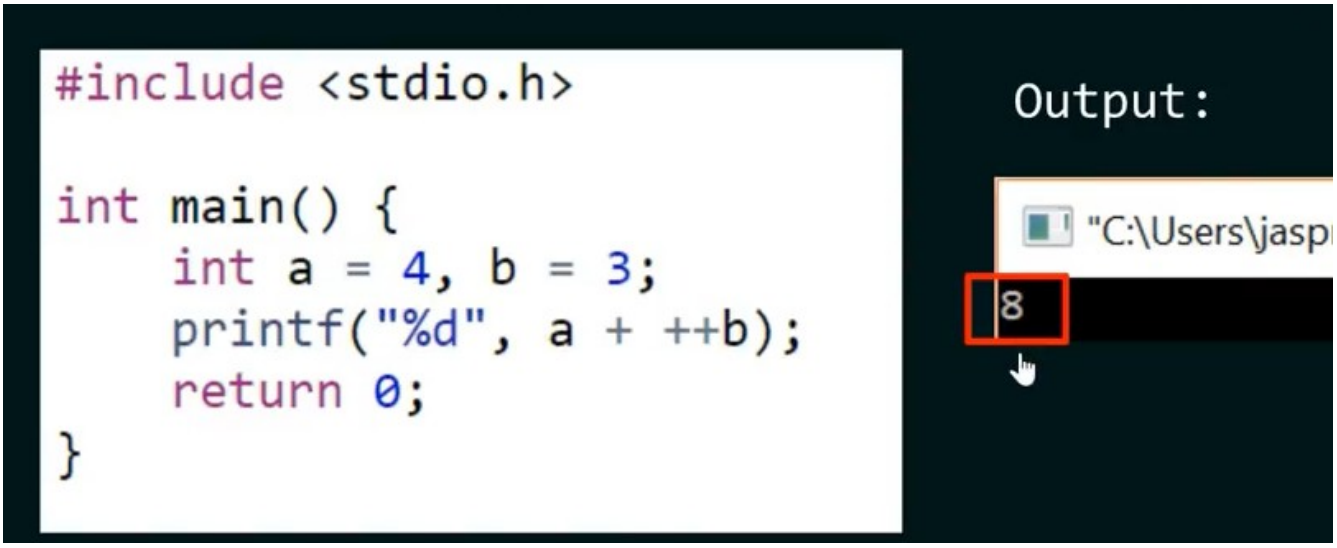





int x = 10 + 5;
Tokens Generated:
int (Keyword)
x (Identifier)
= (Operator)
10 (Integer Literal)
+ (Operator)
5 (Integer Literal)
; (Punctuation)

-------xxxx------


#include <stdio.h>
int main() {
int a = 4, b = 3;
printf("%d", a + ++b);
return 0;
}
Output:
"C:\Users\jasp
8

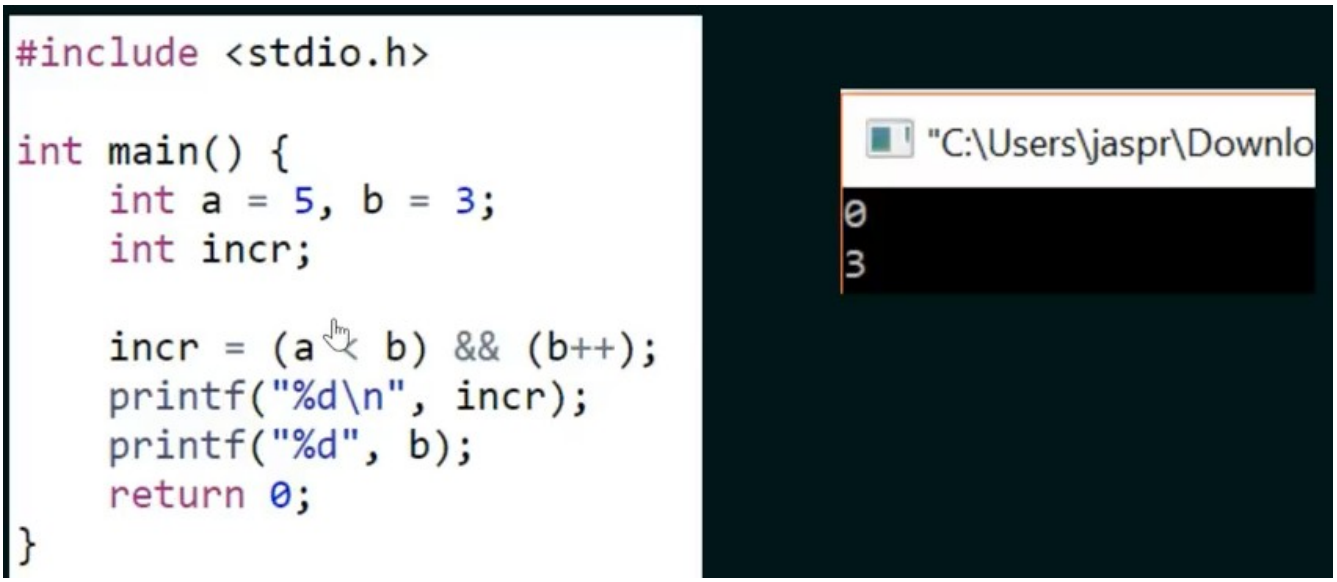


See Lec-24

------x------

---------x--------

short circuit:



***Here in and(&&) logical operator if 1st condition is false 2nd condition will not evaluated or checked.

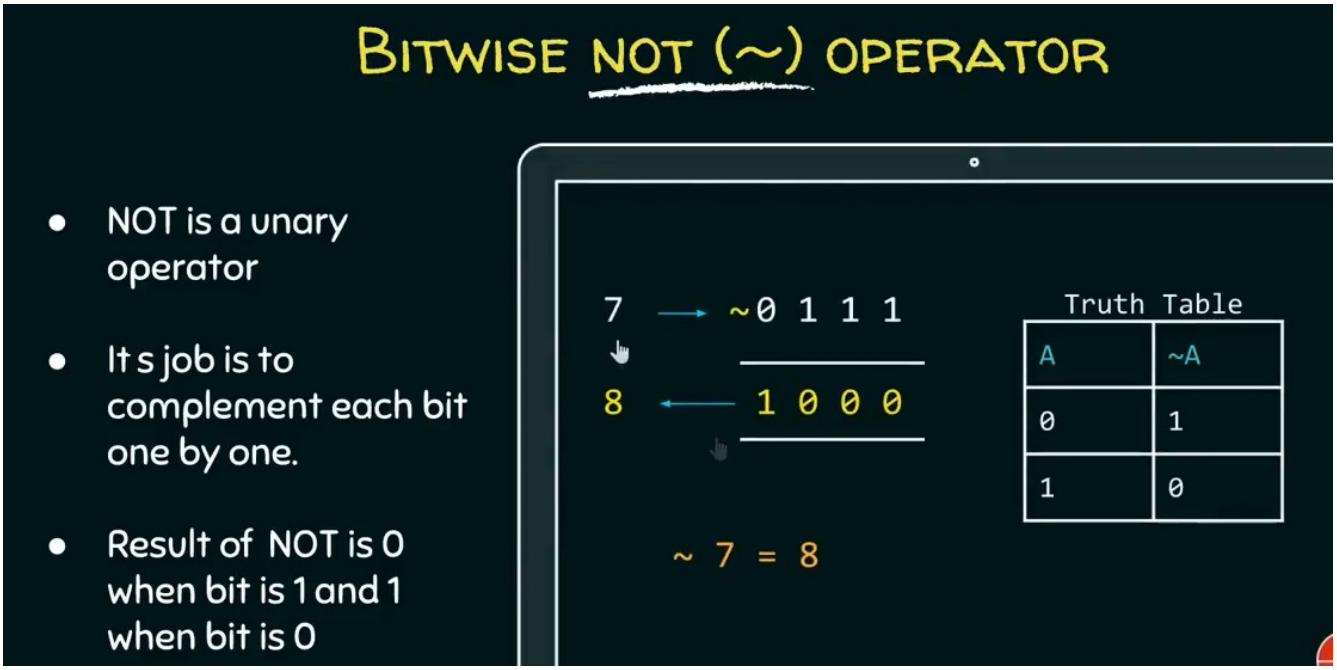
```c
#include <stdio.h>

int main() {
    int a = 5, b = 3;
    int incr;

    incr = (a > b) || (b++);
    printf("%d\n", incr);
    printf("%d", b);
    return 0;
}
```

```
Select "C:\Use
1
3
```

***Here in or(||) logical operator if 1st condition is true 2nd condition will not evaluated or checked.
*** Both called “Concept of **short circuit** in logical operators”

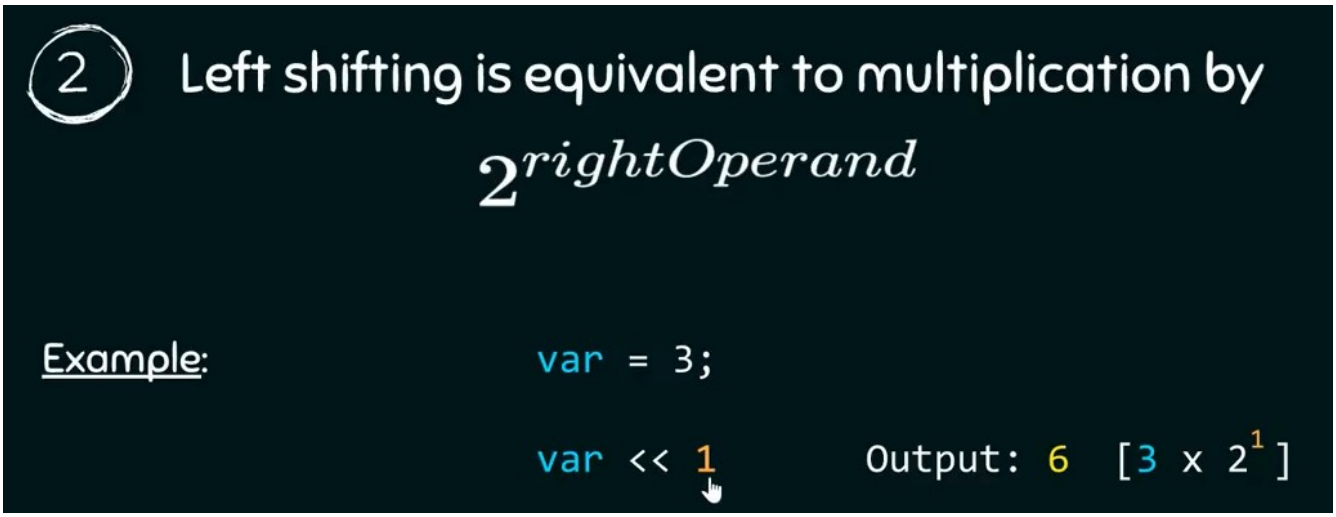
---------x-------

Bitwise operator:
Bitwise NOT(~) operator is unary operator. Means it need 1 operand.

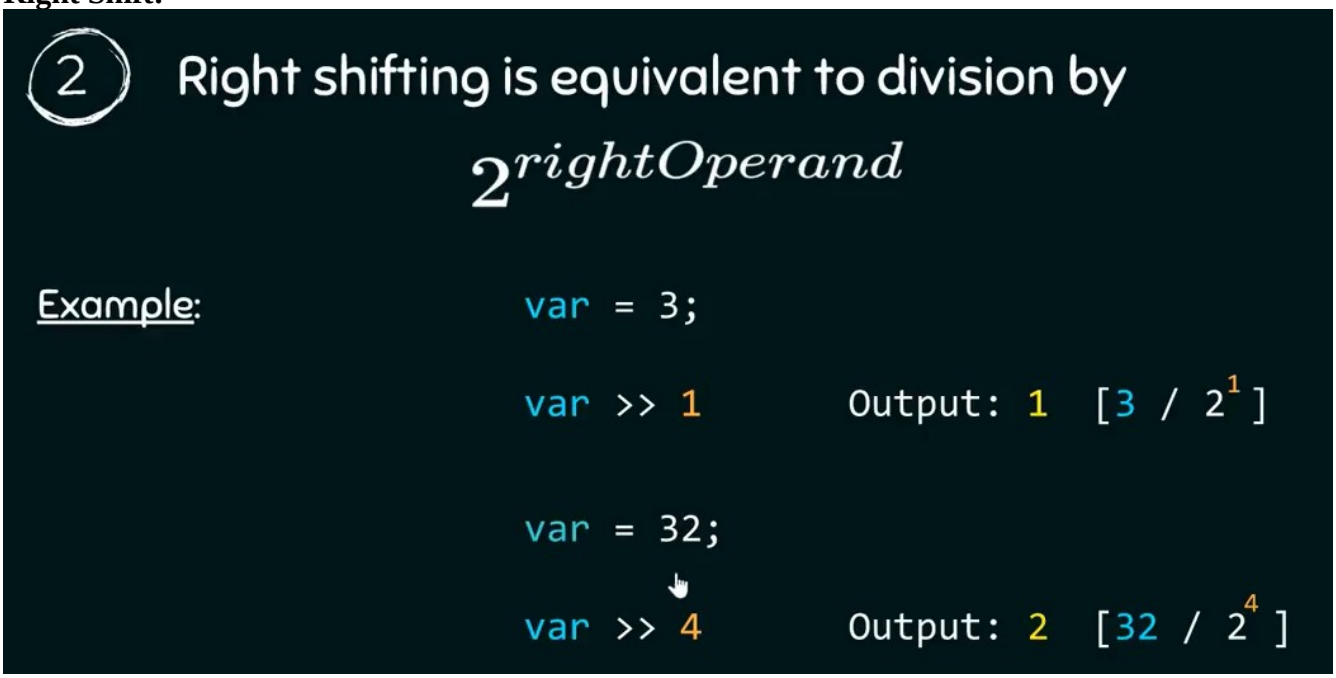


Here Complement means 1 become 0 and 0 becomes 1.

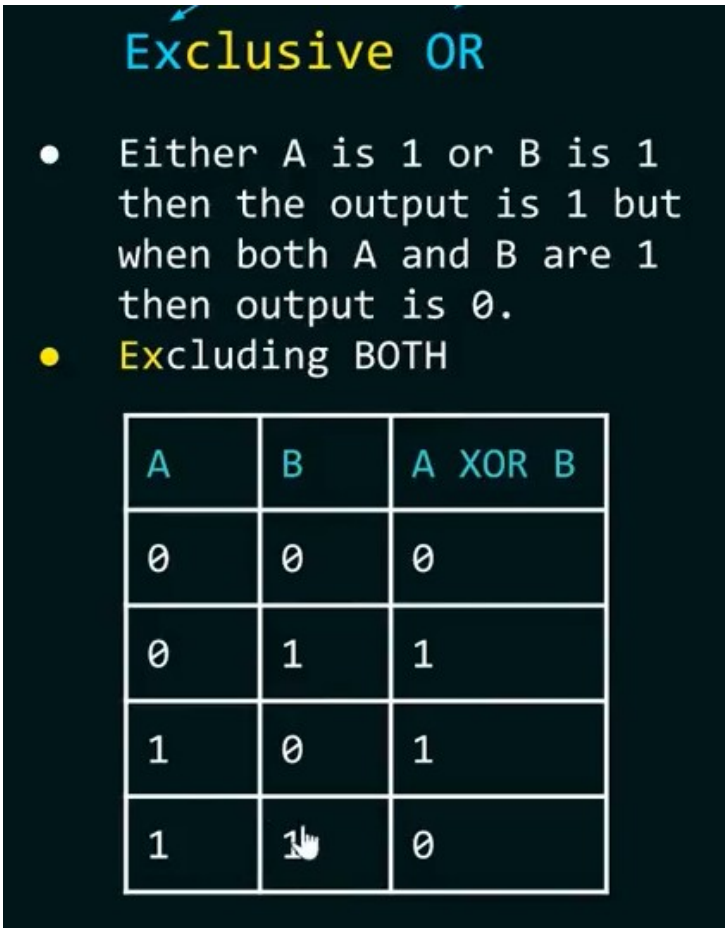
Left Shift:



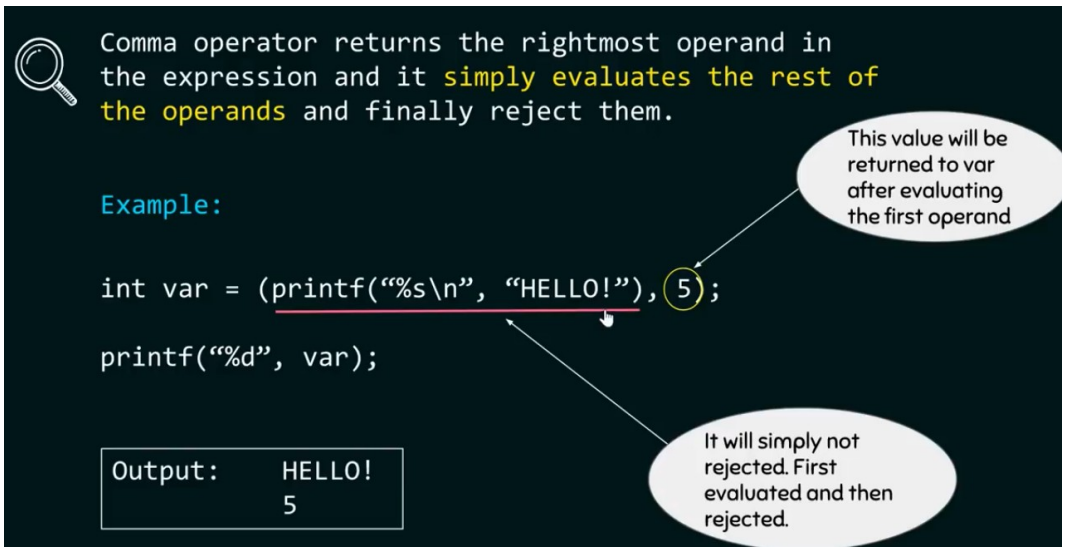
Right Shift:



XOR:



Comma Operator:



Or

```c
#include <stdio.h>

int main(){
    int a=65;
    int var=(3,3,a++,2);
    printf("%d %d", a,var);

}
```

Result will be – a=55, var=2;
Many things to know about comma operator.
For more see lec-33(Comma operator)

----------x---------

Example:

```c
int main()
{
    int a;
    a = fun1() + fun2();
    printf("%d", a);
    return 0;
}

int fun1()
{   printf("Neso");
    return 1;
}

int fun2()
{   printf("Academy");
    return 1;
}
```

Which function is called first ?
fun1() or fun2()?

It is not defined whether fun1() will be called first or whether fun2() will be called. Behaviour is undefined and output is compiler dependent.

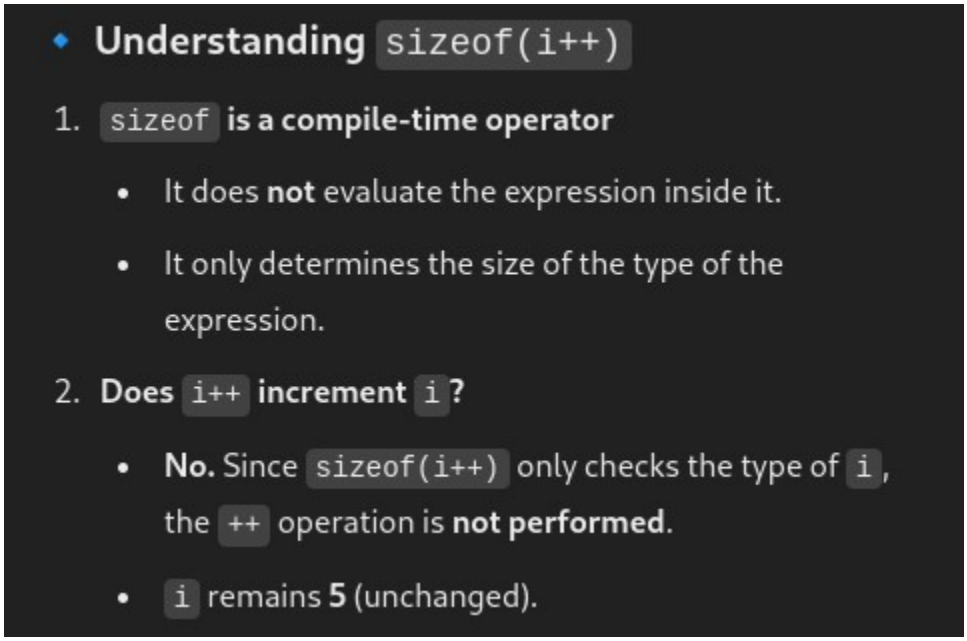
C99 is the older version of C standard adopted in 1999.
C11 is the latest revised version of C standard adopted in 2011.

C Standard: C Standard is the language specification(Rules, Syntex, Feature etc) which is adopted by the all C compiler across the globe. Such as, about the "FOR" loop, something is written inside the standard that explain its functionality, then all C compiler must have to adopt the functionality of "FOR" loop which is provided by C standard.

QUIZ:

```c
#include <stdio.h>
int main()
{
    int i = 5;
    int var = sizeof(i++);
    printf("%d %d", i, var);
    return 0;
}
```

Ans: i=5, var=4;
Here 'i++' will not evaluated. Because 'sizeof' is compile time operator. It only determines the size of the type. **See Lec-35**(Operator in C, solved prblm 1);



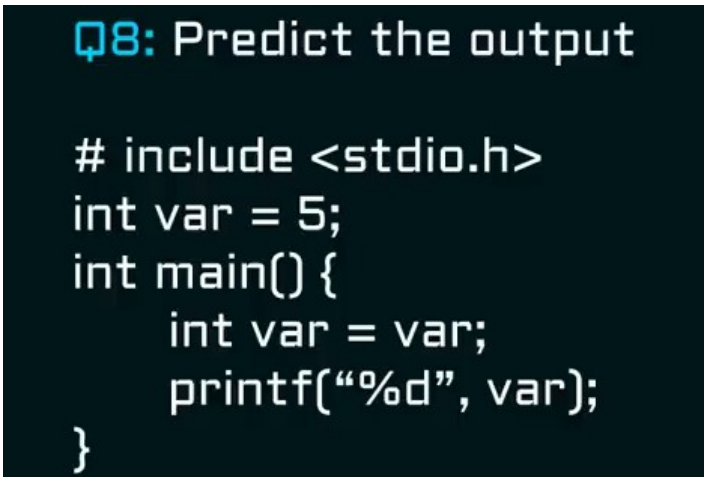
-----x-----

**Post-increment Means after the completion of the expression increment will be evaluated.

```c
#include <stdio.h>
int main() {
    int c=9;
    printf("%d", c++);
    return 0;
}
```

Here ans will c=9;
after expression completed c will increment to 10.

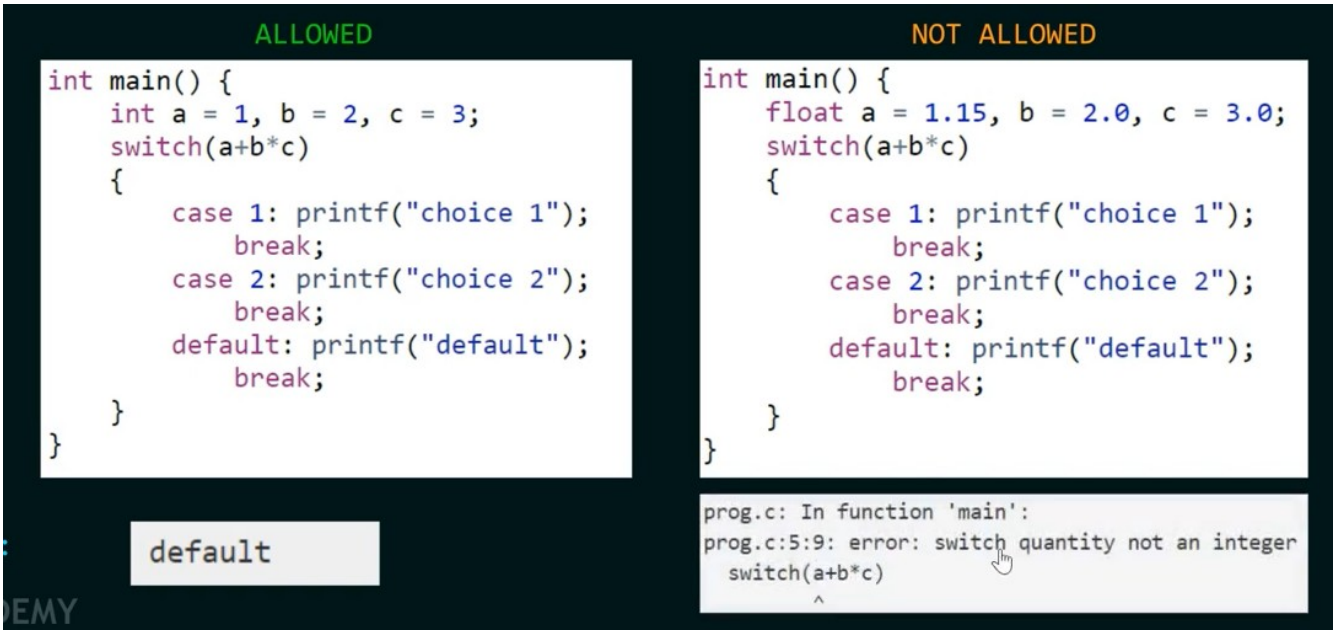
QUIZ:



Ans will be **garbage value**. Because local and global variable has same name, so local variable is shadowing global variable. Wrong Initialization.

----------x---------

Switch: Switch is great replacement of long **if-else** construct.
***Default** case is optional in switch statement.



NOT ALLOWED

```
int main() {
    float x = 3.14;
    switch(x)
    {
        case 3.14: printf("x is 3.14");
            break;
        case 1.1: printf("x is 1.14");
            break;
        case 2: printf("x is 2");
            break;
    }
}
```

```
prog.c:7:6: error: case label does not reduce to an integer constant
     case 3.14: printf("x is 3.14");
     ^
prog.c:9:6: error: case label does not reduce to an integer constant
     case 1.1: printf("x is 1.14");
```

ALLOWED

```
int main() {
    int x = 23;
    switch(x)
    {
        case 3+3: printf("choice 1");
            break;
        case 3+4*5: printf("choice 2");
            break;
        default: printf("default");
            break;
    }
}
```

```
choice 2
```

4 Variable expressions are not allowed in case labels. Although macros are allowed.

```
#include <stdio.h>
#define y 2
#define z 23

int main() {
    int x = 2;
    switch(x)
    {
        case y: printf("Number is 2");
            break;
        case z: printf("Number is 23");
            break;
        default: printf("default case");
            break;
    }
}
```

Output:

```
Number is 2
```

5 Default can be placed anywhere inside switch. It will still get evaluated if no match is found.

```
int main() {
    int x = 2, y = 2, z = 23;
    switch(x)
    {
        case y: printf("Number is 2");
            break;
        case z: printf("Number is 23");
            break;
        default: printf("default case");
            break;
    }
}
```

```
prog.c:7:6: error: case label does not reduce to an integer constant
      case y: printf("Number is 2");
      ^
prog.c:9:6: error: case label does not reduce to an integer constant
      case z: printf("Number is 23");
```

CADEMY

```
int main(){
    int i=0;
    for(i=0; i<20; i++){
        switch(i){
            case 0: i+=5;
            case 1: i+=2;
            case 5: i+=5;
            default: i+=4;
        }
        cout<<i<<" ";
    }
```

***** If **"Break"** statement is not used, then if one statement is matched, all subsequent cases will be executed until end of switch case. Here, on 1st loop case-0,1,5,default will execute, on 2nd loop only default will execute.

----------x--------

Binary To Decimal:

```
printf("Enter the binary number: ");
scanf("%d", &binary);

decimal = 0, weight = 1;

while(binary != 0)
{
    rem = binary % 10;
    decimal = decimal + rem*weight;
    binary = binary / 10;
    weight = weight*2;
}
```

```
1 0 0 1

rem = 1
decimal = 1
binary = 1 0 0
weight = 1*2 = 2
```

Power of an Int: For Negaive Power(2^{-3}). For Positive exponent find what to change.

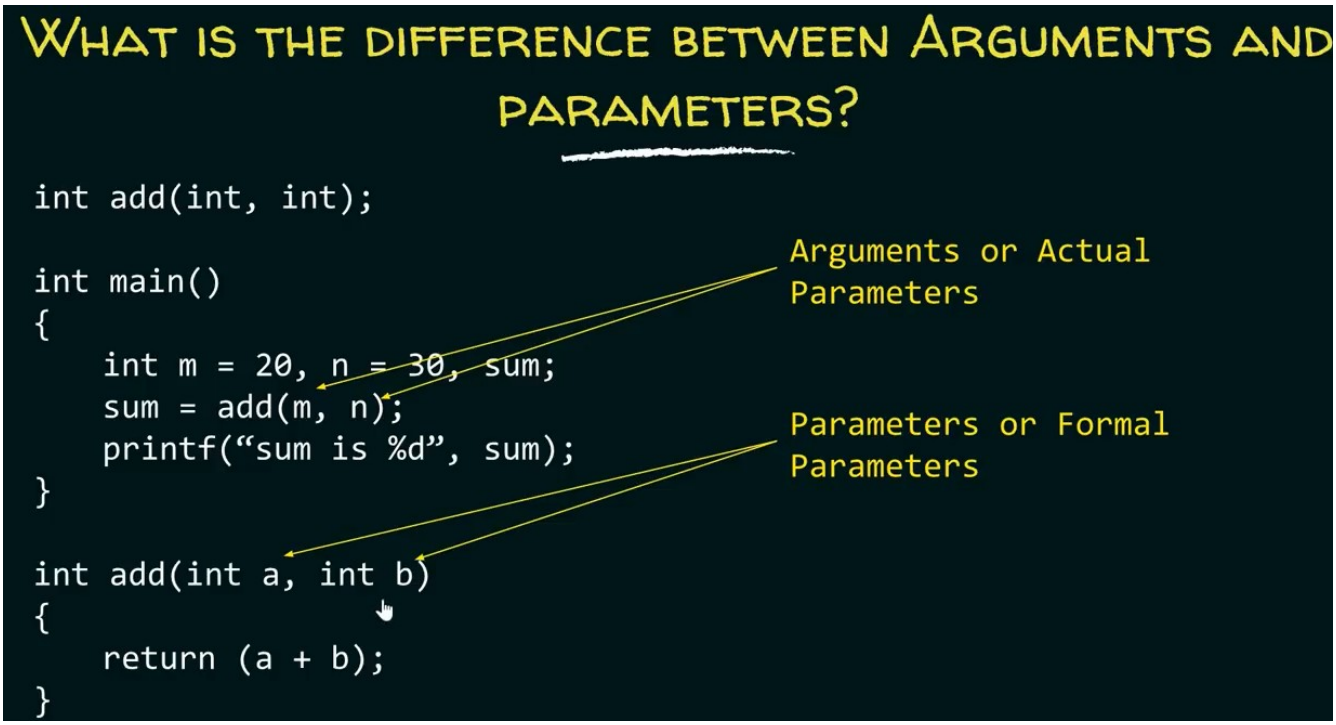
IMPLEMENTATION

```
int base, exponent, expo;
double power1=1.0;
printf("Enter the base: ");
scanf("%d", &base);
printf("Enter the exponent: ");
scanf("%d", &exponent);

expo = exponent;
if(exponent < 0)
{
   while(exponent!=0)
   {
       power1 = power1*(1.0/base);
       exponent++;
   }
   printf("%d to the power of %d is %.10f", base, expo, power1);
```

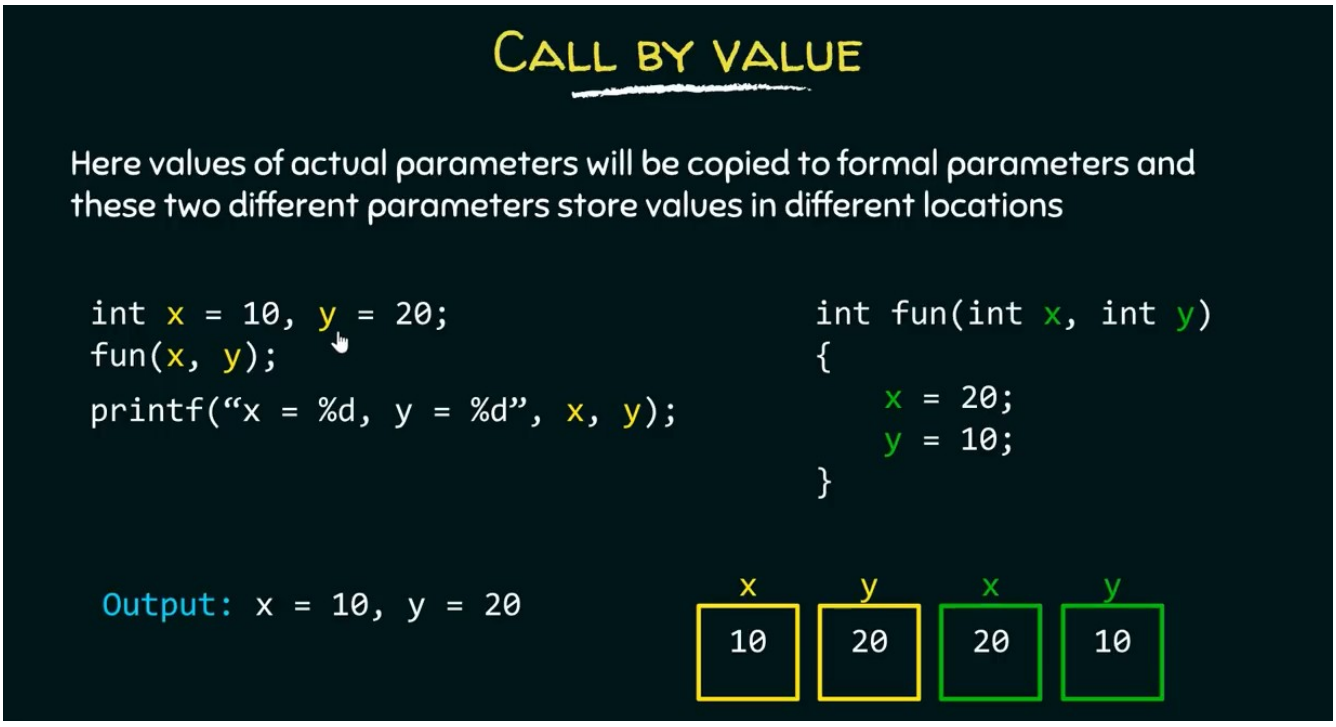
exponent
-3
2
base

```
Iteration 1:
power1 = 1*(1.0/2) = 0.5
exponent =  -3+1 = -2
```

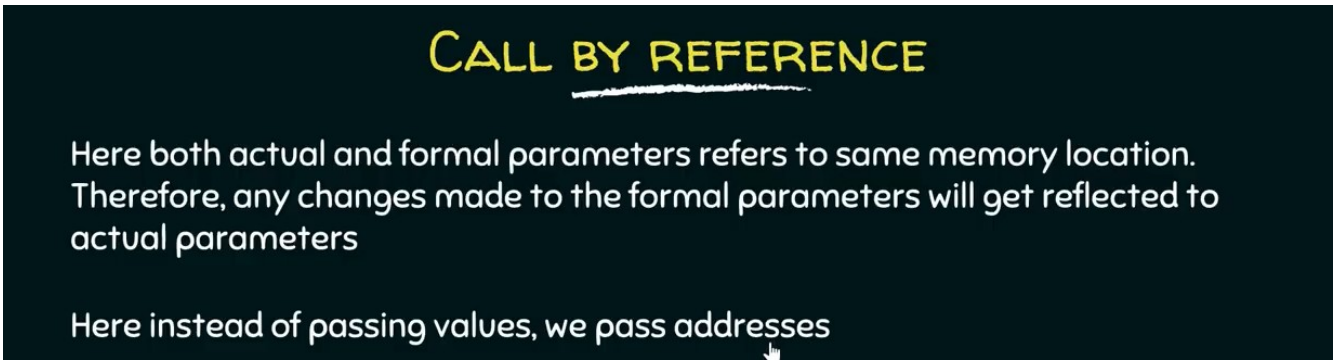
Function: In C, Function are global by default.



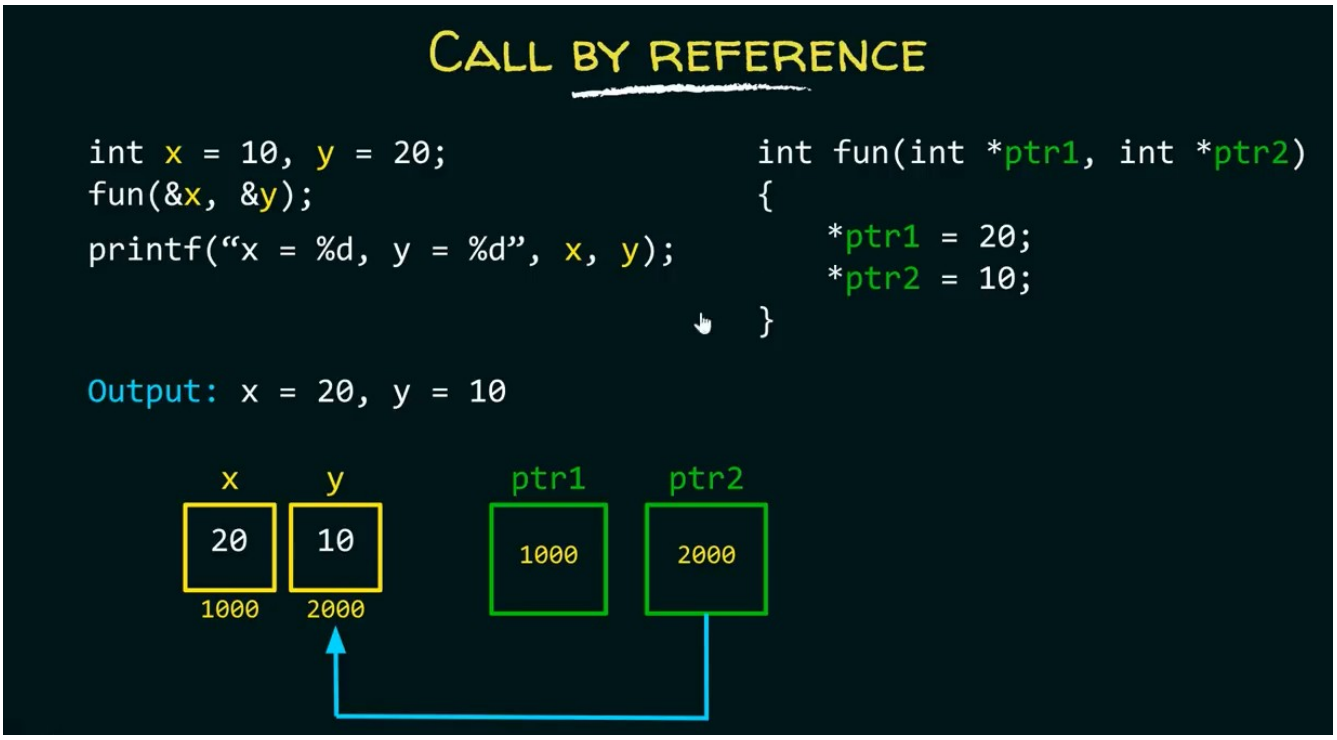
Call By Value:



Call By reference:

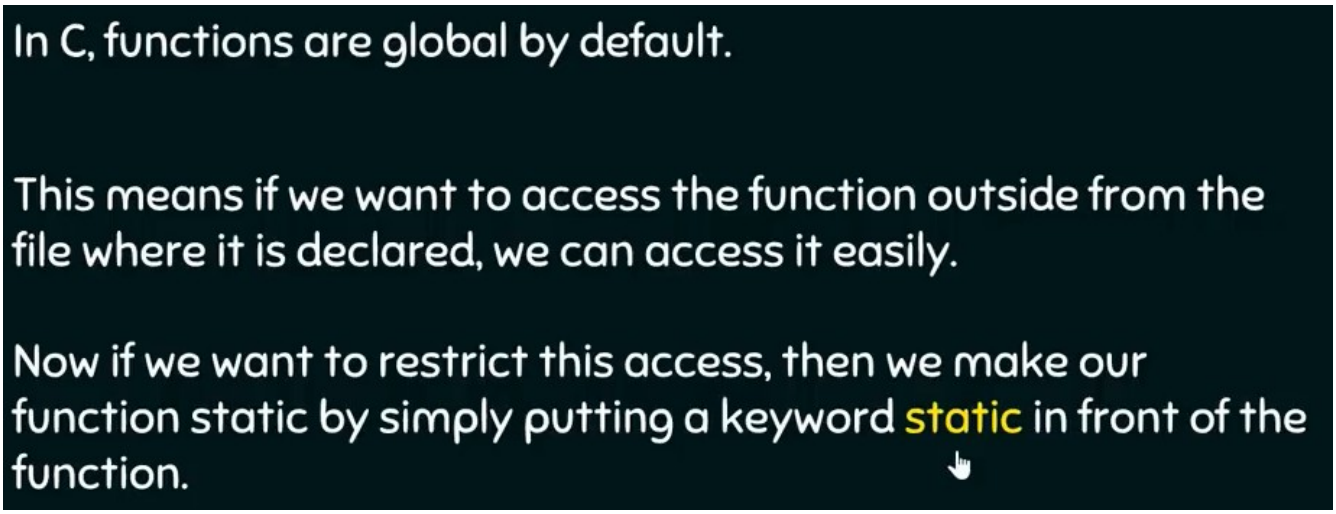


*** &x and *ptr here '&(reference Operator)' referencing value, '*(De-reference Operator)' is de-referencing value. They are opposite to each other.



----------x----------

Static Function: Static functions are restricted to their file where they are declared. No other file can't access these functions.



See Lec-68(Static Function) for details understanding

---------x-------

Static & Dynamic Scoping: C lang follows static scoping. To know details about Dynamic scoping See Lec-69-71(Static & Dynamic scoping).

If a variable is not found so that scope, then it will search outer containing block or scope. Like

```
int a=5;

int main(){
    int a=10;
    if(1){
        int b=10;
         b=b+a;
    }
}
```

Here in 'if' statement a=10 will count. If a=10 if comment out then a=5 will count.

Below Example…

```
int fun1(int);
int fun2(int);
int a = 5;
int main()
{
    int a = 10;
    a = fun1(a);
    printf("%d", a);
}

int fun1(int b)
{
    b = b+10;
    b = fun2(b);
    return b;
}

int fun2(int b)
{
    int c;
    c = a + b;
    return c;
}

Output: 25
```

SO ACADEMY

This ans is for Static Scoping.

Here in 'fun2' 'a' variable is not found on this scope. It will search for 'a' outer scope. So a=5;

But in case of **Dynamic scoping**

In dynamic scoping, definition of a variable is resolved by searching its containing block and if not found, then searching its calling function and if still not found then the function which called that calling function will be searched and so on.

DYNAMIC SCOPING EXAMPLE

```
int fun1(int);
int fun2(int);
int a = 5;
int main()
{
    int a = 10;
    a = fun1(a);
    printf("%d", a);
}

int fun1(int b)
{
    b = b+10;
    b = fun2(b);
    return b;
}

int fun2(int b)
{
    int c;
    c = a + b;
    return c;
}

Output: 30
```

SO ACADEMY

In ‘fun2’ ‘a’ variable is not found, so it will search on callee function(fun1) for ‘a’. if not found it search again for that callee function(main) and use a=10; if not found use global variable a=5;

Thats why for static scoping ans=25;

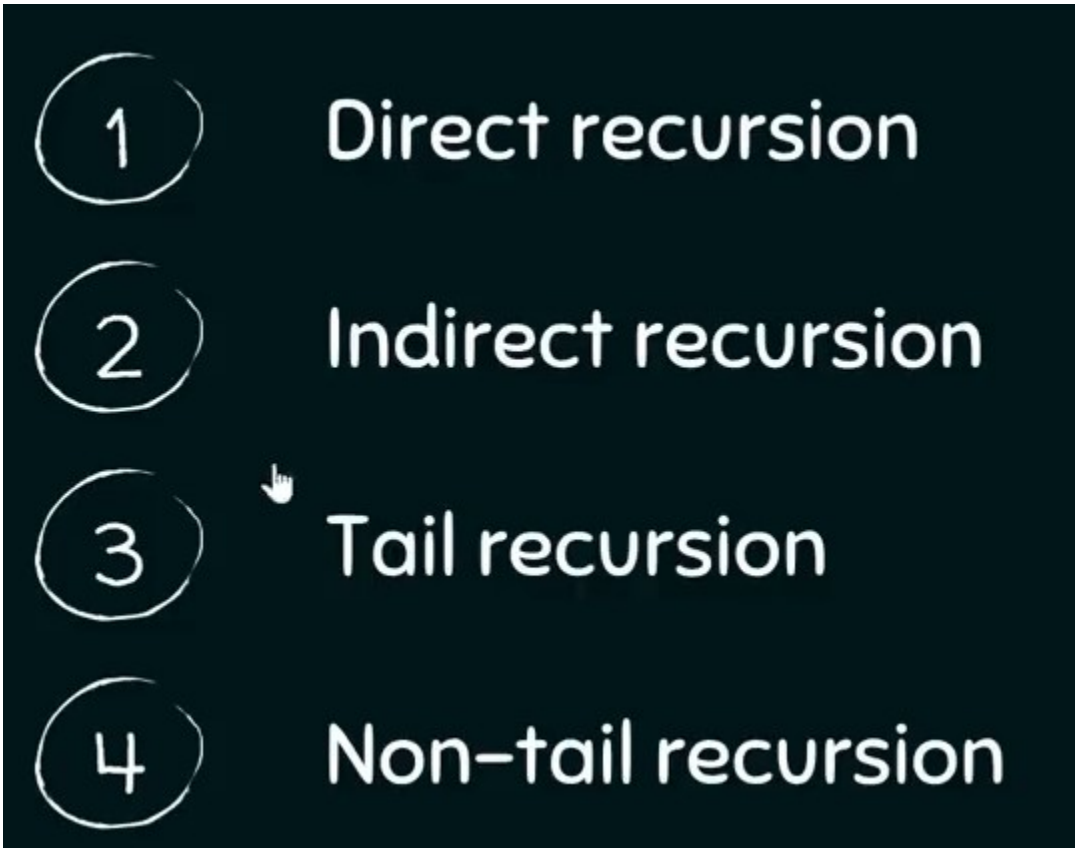
for Dynamic scoping ans=30;

---------x-------

Recursion:

* Must specify base condition to stop the recursion.

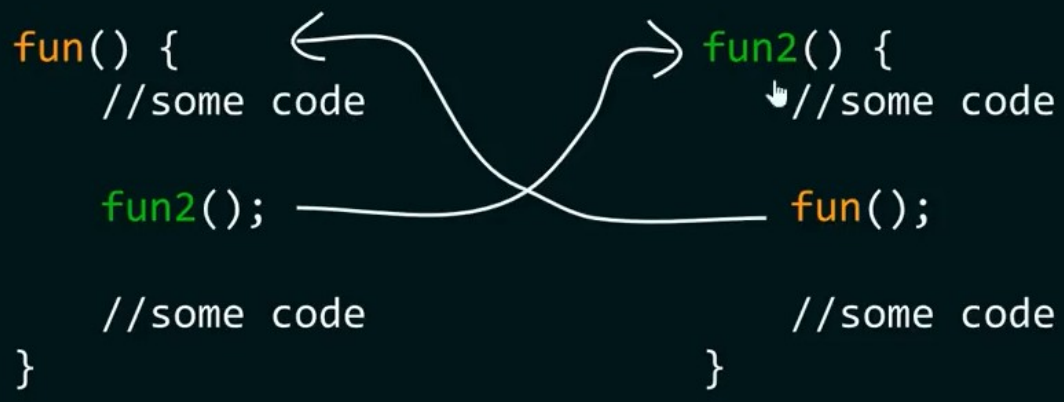
Type of Recursion:



Indirect recursion

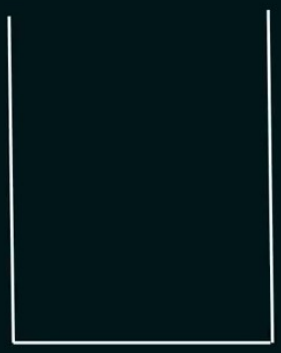
A function (let say fun) is called indirect recursive if it calls another function (let say fun2) and then fun2 calls fun directly or indirectly.

Structure of Indirect recursion:



A recursive function is said to be tail recursive if the recursive call is the last thing done by the function. There is no need to keep record of the previous state.

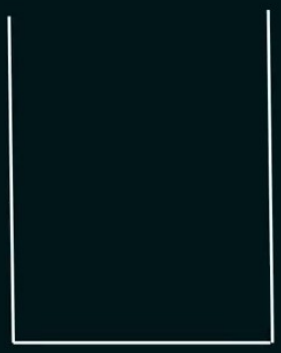
```
void fun(int n) {
    if(n == 0)
        return;
    else
        printf("%d ", n);
    return fun(n-1);
}
int main() {
    fun(3);
    return 0;
}
```



Output: 3 2 1

A recursive function is said to be non-tail recursive if the recursive call is not the last thing done by the function. After returning back, there is some something left to evaluate.

```
void fun(int n) {
    if(n == 0)
        return;
    fun(n-1);
    printf("%d ", n);
}
int main() {
    fun(3);
    return 0;
}
```



Output: 1 2 3

Advantage and Disadvantage: Every Recursive program can be written as iterative program.

* Recursive programs are more elegant and requires less lines of code

* Recursive program requires more space(Stack Space) than iterative programs. Because in recursion for every function call there is a activation record stored on call stack. But no function are stored on iterative..

Const Array: Can't change ant element of an array. Const a[3]={1,2,3};

Pointer

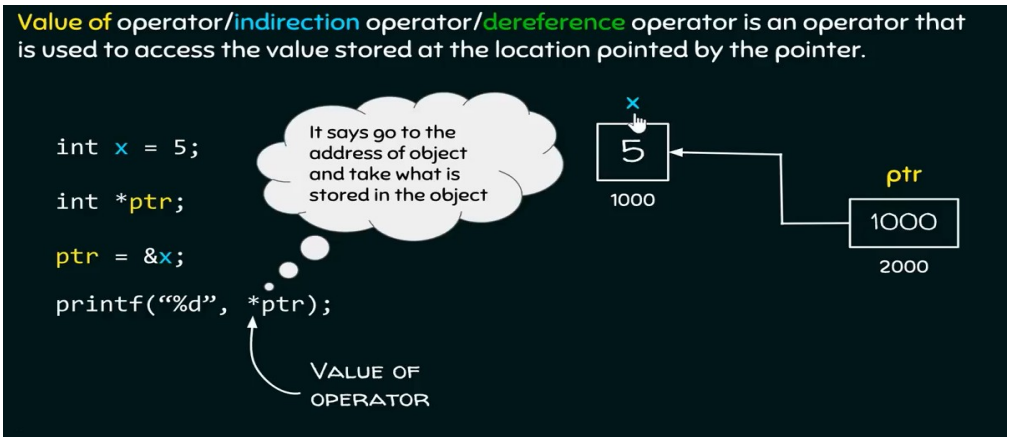
Pointer is a special variable which is capable of storing the initial address of an object which it wants to point. Simply point to the base address of the object. Variable **int i=6**; address={101, 102, 103, 104}. pointer will point(store address) to 101 address.

int x; // declaring variable.

X=5; // initializing variable

int *ptr=&x; // pointer **ptr** is pointing to variable **x**.

Dereference Operator:



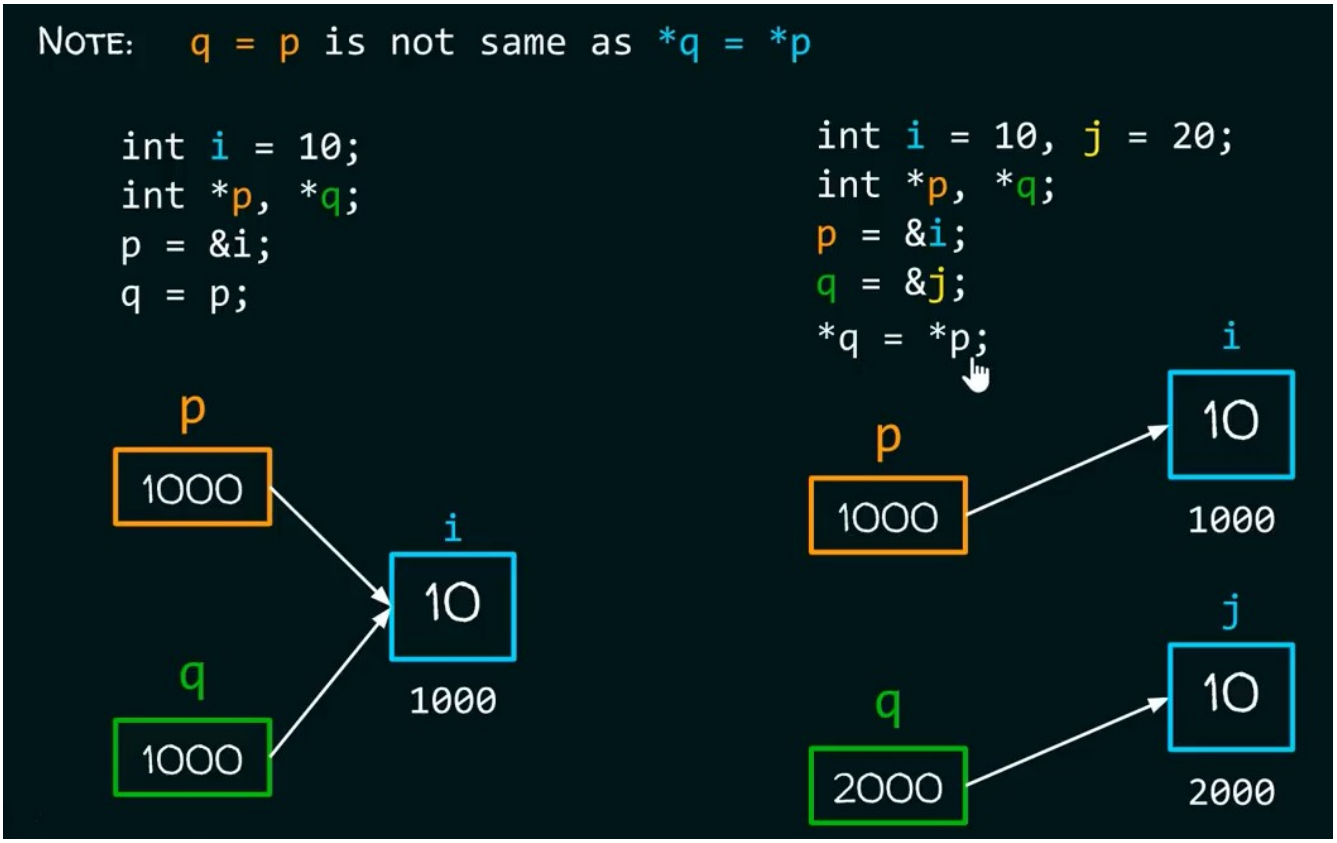
Can change variable value with pointer.

Int x=5, *ptr=&x;

*prt=4; // **x** value will be 4.

```c
int x=6;
int *ptr;
ptr=&x;
*ptr=5; // while changing variable value '*' have to use
printf("%d", *ptr);
```

*Important



Herejvalue changes to 10 from 20.

****Important**:

```c
#include <stdio.h>

void minMax(int arr[], int len, int *min, int *max)
{
    *min = *max = arr[0];
    int i;
    for(i=1; i<len; i++)
    {
        if(arr[i] > *max)
            *max = arr[i];
        if(arr[i] < *min)
            *min = arr[i];
    }
}

int main() {
    int a[] = {23, 4, 21, 98, 987, 45, 32, 10, 123, 986, 50, 3, 4, 5};
    int min, max;
    int len = sizeof(a)/sizeof(a[0]);
    minMax(a, len, &min, &max);
    printf("Minimum value in the array is: %d and Maximum value is: %d", min, max);
    return 0;
}
```

See “Lec:- 106-Pointer application”

Return pointer(address): Finding array mid value by pointer

```c
int main()
{
    int a[] = {1, 2, 3, 4, 5};
    int n = sizeof(a)/sizeof(a[0]);
    int *mid = findMid(a, n);
    printf("%d", *mid);
    return 0;

}

int *findMid(int a[], int n)
{
    return &a[n/2];
}
```

****important:** automatic variable means local variable

Never ever try to return the address of an automatic variable

For example:

```
int *fun()
{
    int i=10;
    return &i;
}

int main() {
    int *p = fun();
    printf("%d", *p);
}
```

```
Warning: function returns address
of local variable
```

Question 1: Consider the following two statements

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

First statement is the declaration and second is simple assignment statement. Why isn't in second statement, p is preceded by * symbol?

Solution: In C, * symbol has different meanings depending on the context in which it's used.

At the time of declaration, * symbol is not acting as an indirection operator.

* symbol in the first statement tells the compiler that p is a pointer to an integer.

But, if we write *p = &i then it is wrong, because here * symbol indicates the indirection operator and we cannot assign the address to some integer variable.

Therefore, in the second statement, there is no need of * symbol in front of p. It simply means we are assigning the address to a pointer.

Printing address of a pointer. **Printf("%p", ptr);

Question 4: If i is a variable and p points to i, which of the following expressions are aliases of i?

a) *p b) *&p c) &p d) *i e) *&i

Solution: Options (a) and (e) are the aliases of variable i

Here **“*&i” “*p”** gives the same value.

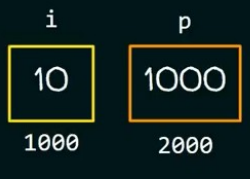
Question 4: If i is a variable and p points to i, which of the following expressions are aliases of i?

a) *p b) *&p c) &p d) *i e) *&i

Solution: Options (a) and (e) are the aliases of variable i

Example

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



a) *p = *(1000) = 10

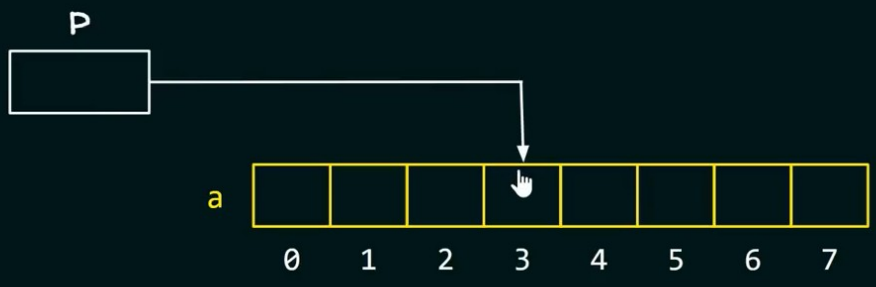
b) *&p = *(&p) = *(2000) = 1000

c) &p = 2000

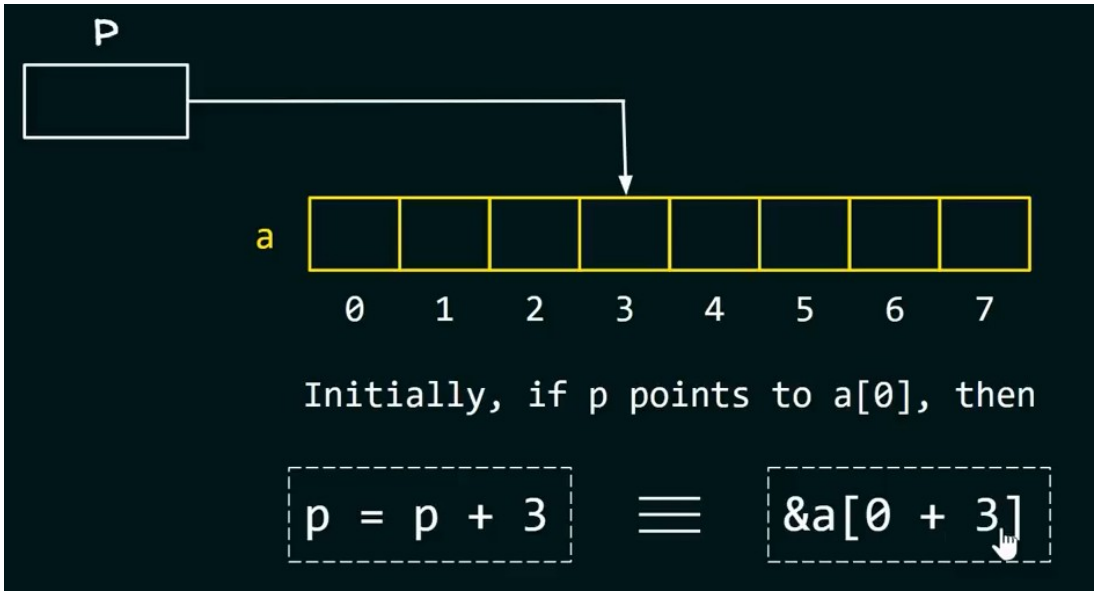
d) *i = *(10) doesn’t make sense

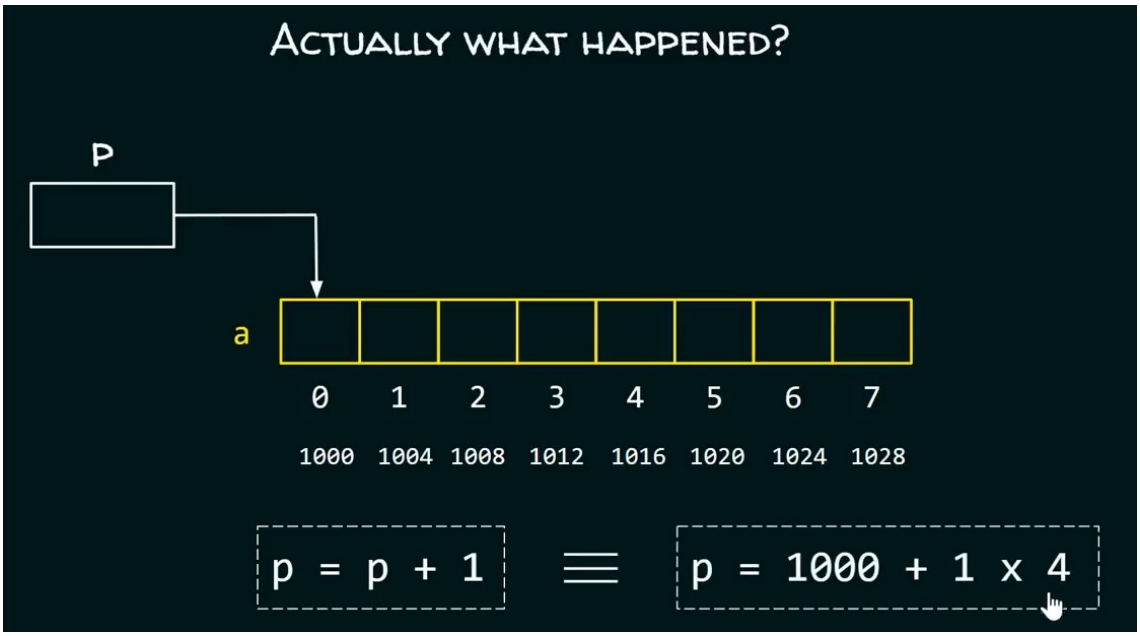
e) *&i = *(&i) = *(1000) = 10

WHAT HAPPENS IF WE ADD SOME INTEGER TO THE POINTER?



`p = p + 3` means moving the pointer 3 positions in forward direction.





***Same rule for subtraction.

```c
int main() {
    int a[] = {5, 16, 7, 89, 45, 32, 23, 10};
    int *p = &a[0];
    printf("%d ", *(p++));
    printf("%d", *p);
    return 0;
}
```

Ans: 5 16

******* Compare the results…

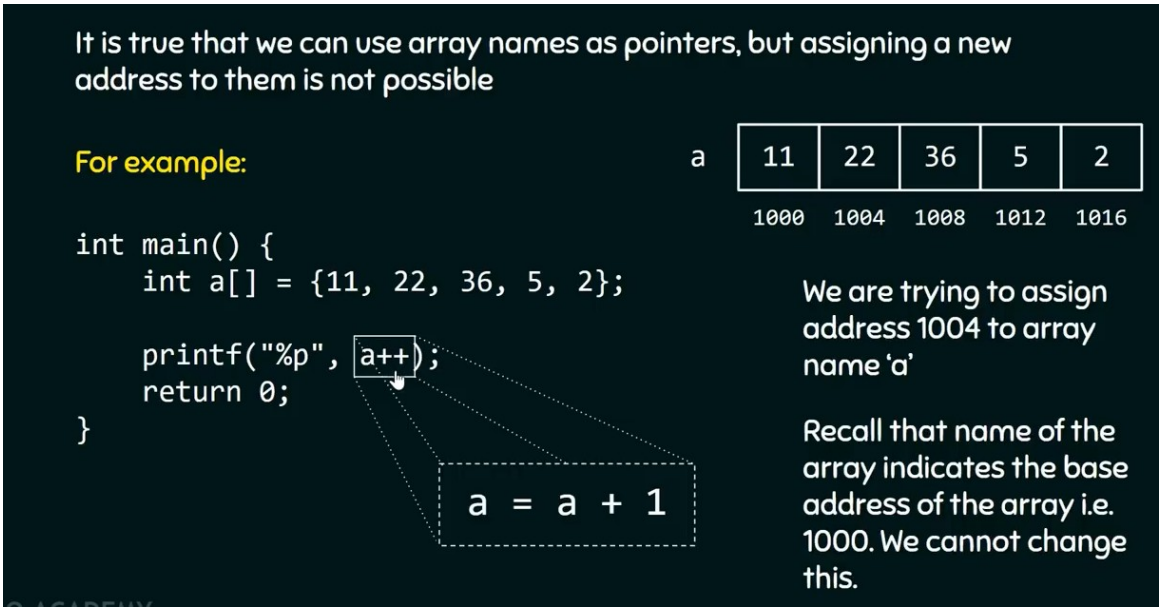
```c
int a[8]={5,16,7,89,45,32,23,10};
int *p=&a[1], *q=&a[5];
printf("%d ", *(p++)); //ans: 16
printf("%d\n", *(q++)); //ans: 32
printf("%d ", *(p+3)); //ans: 32
printf("%d\n", *(q-3)); //ans: 89
printf("%d ", *p); //ans: 7
printf("%d\n", *q); //ans: 23
printf("%d ", q-p); //ans: 4
printf("%d ", p<q); //ans: 1
printf("%d ", *p<*q); //ans: 1
```

Remember: If using "++" pointer will change its location, but using "+ or -" does not changes location.

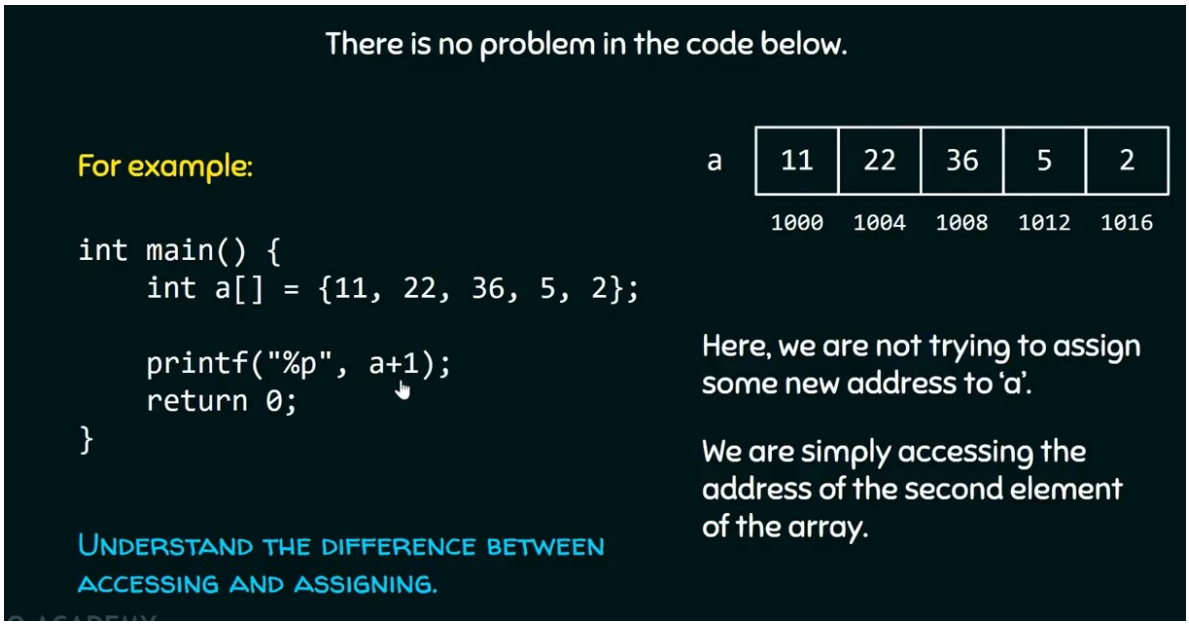
FACT: NAME OF AN ARRAY CAN BE USED AS A POINTER TO THE FIRST ELEMENT OF AN ARRAY.

For example:

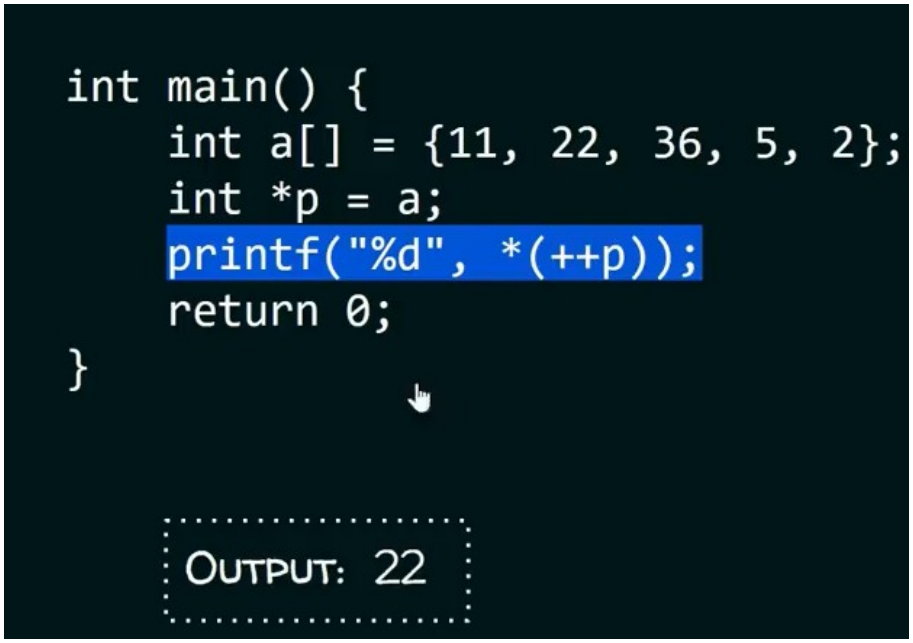
```c
int main() {
    int a[5];
    *a = 10;
    printf("%d", a[0]);
    return 0;
}
```



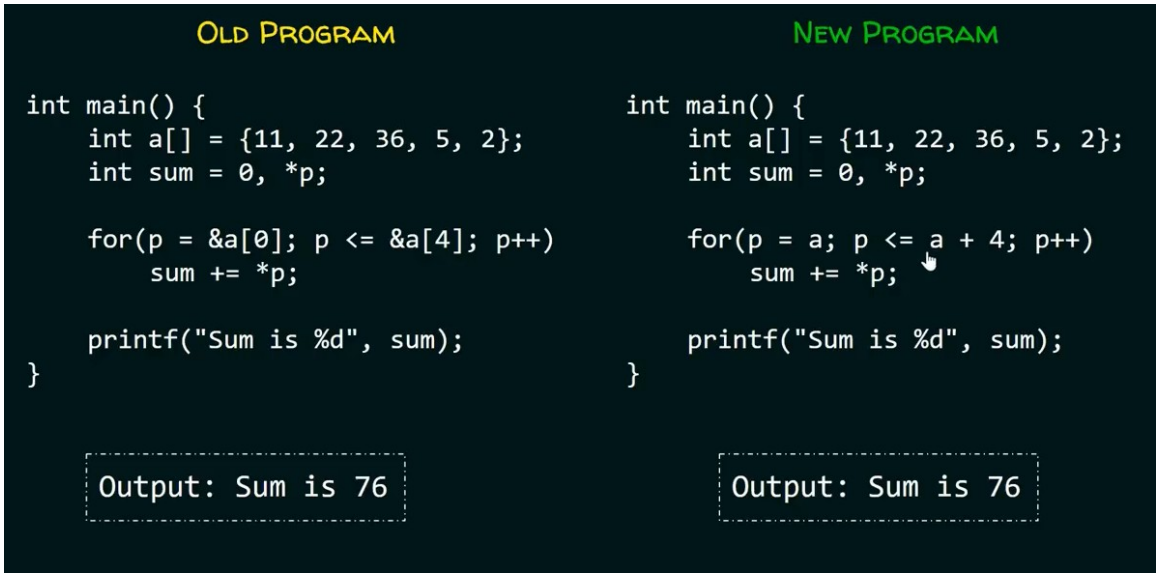
This is not allowed. Because we are changing the base address from 1000 to 1004. Instead of this we can write below code.



Alternative: Assigning the base address to new pointer.



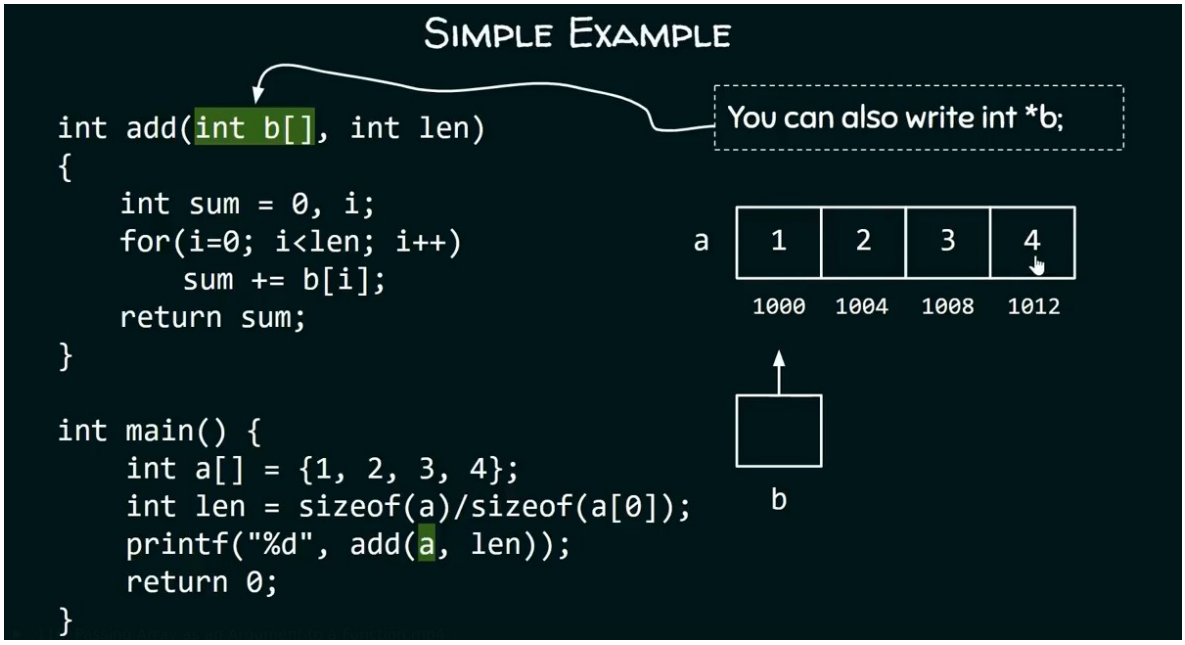
----------x--------



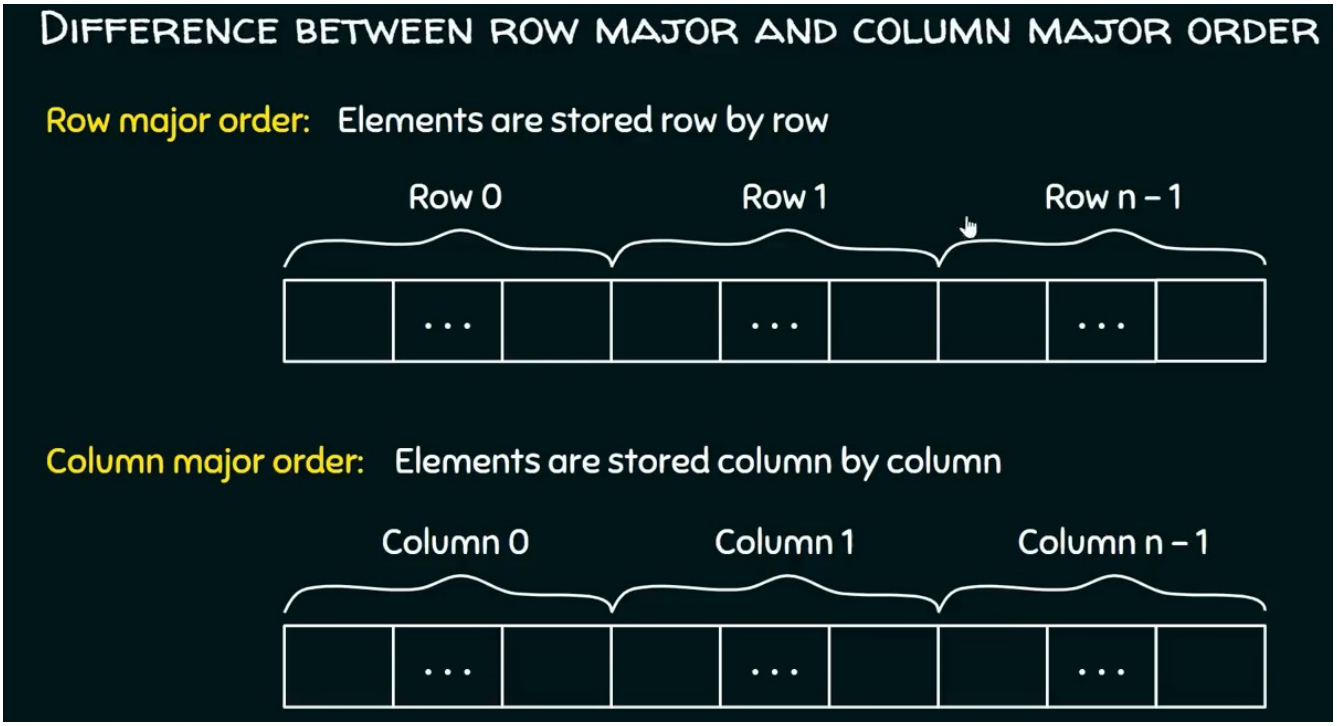
*Print array reverse order by pointer:

```c
#include <stdio.h>
#define N 5

int main() {
    int a[N], *p;
    printf("Enter 5 elements in the array: ", N);
    for(p=a; p <= a+N-1; p++)
        scanf("%d", p);

    printf("Elements in reverse order:\n");
    for(p = a+N-1; p >= a; p--)
        printf("%d ", *p);

    return 0;
}
```

*while passing an array to a function we actually pass the base address to the function.

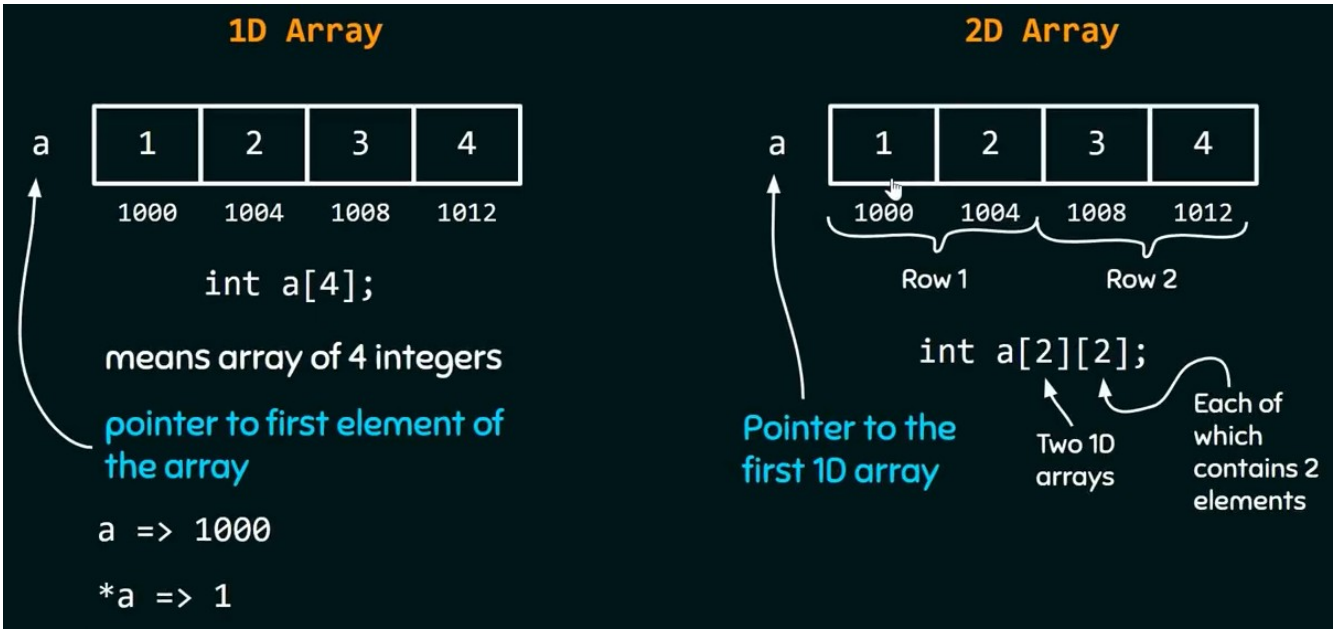


Row/ Column major order:



***C/C++ follows row major order.

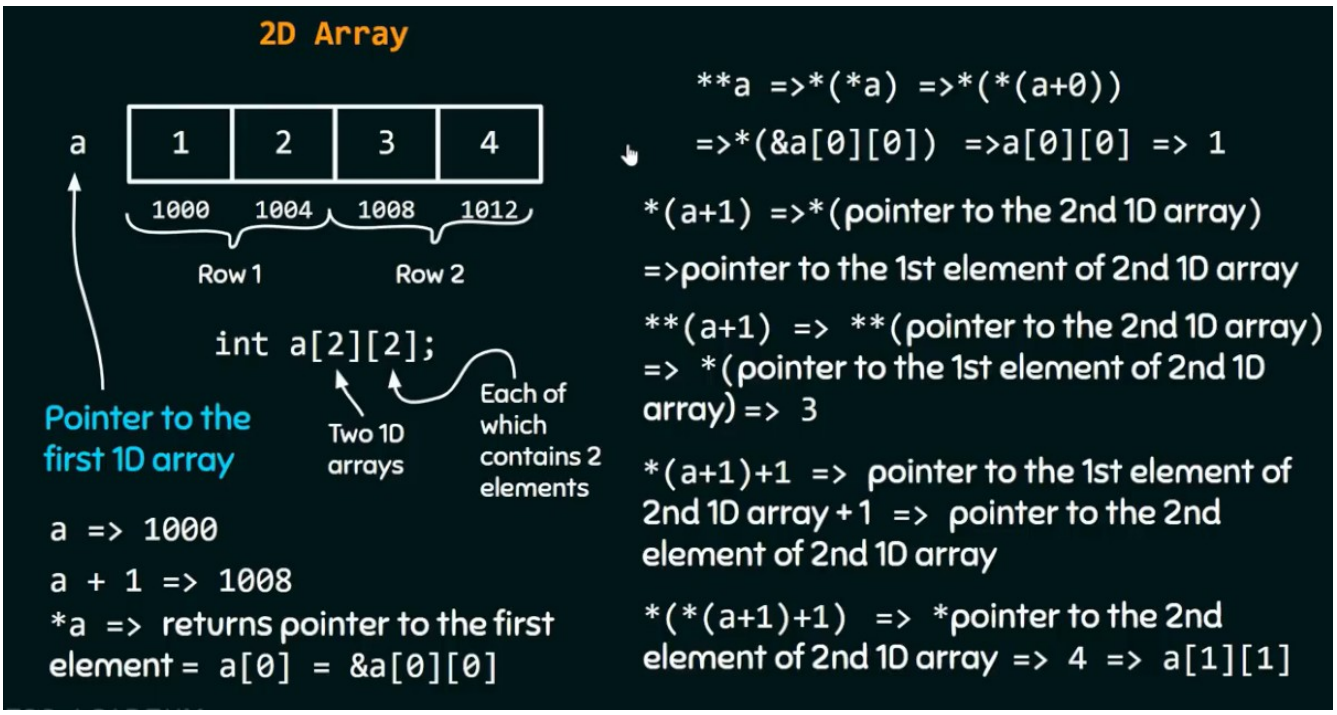
*****Array & Multi-dimensional Array in terms of pointer:**

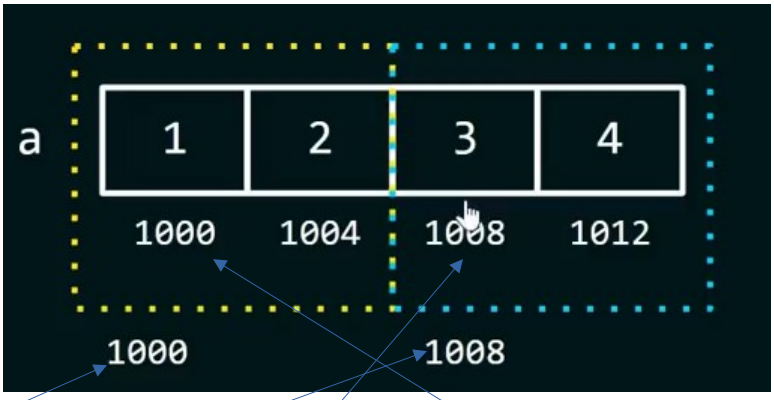


int a[2][2]; Means ‘a’ is 2D array, it has 2 one dimensional array, each array has 2 elements.

In **1D** array, array ‘a’ points to the first element of the array, But in multi-dimensional(**2D, 3D, 4D**) array array ‘a’ points to the 1st row(or 1st 1D array) of the array, not the 1st element, but points on all elements of the 1st row.

**See lec-118(Processing the multidimensional array elements)





This is 2D array. if we write ‘a’ it points to the 1st 1D array.

a=> 1000 (pointer to 1st 1D array)

a+1=> 1008 (pointer to 2nd 1D array)

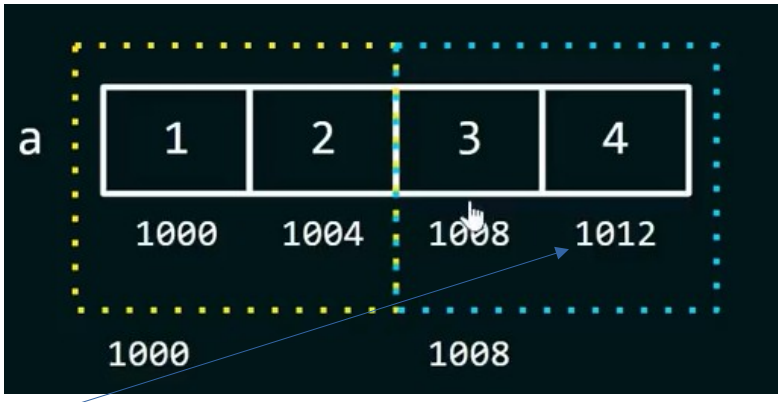
*a=> *(pointer to 1st 1D array) // putting ‘*’ means get inside 1D array, and get 1000 as address as the 1st element of the array. So we can write>> ***a=> pointer to the 1st element of the 1st 1D array***. Contains address of 1st element of the 1st 1D array.

Or we can write *a=> *(a+0) => a[0]=> &a[0][0];

a=> value of the 1st element of the 1st 1D array. // dereferencing, get inside and grab the value of that address. So ****a=> 1

*(a+1)=> 1008(pointer to the 1st element of the 2nd 1D array.)

**(a+1)=> 3



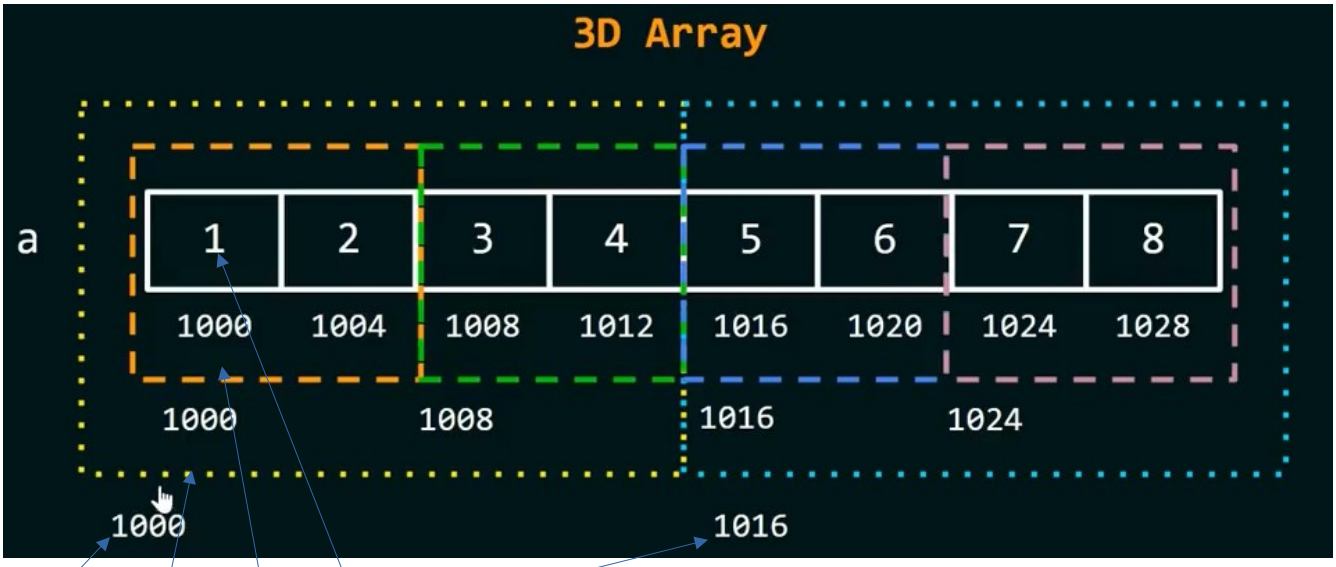
*(a+1)+1=> 1012

*(*a+1)=>2

((a+1)+1) or **((a+1)+1)=> 4;

a[2][2][2]=> means ‘a’ is 3D array. It has 2 two dimensional array. Both 2D array has 2 one dimensional array. Both 1D array has 2 elements.

3D array:



a=>1000;

a+1=> 1016;

*a=>1000;

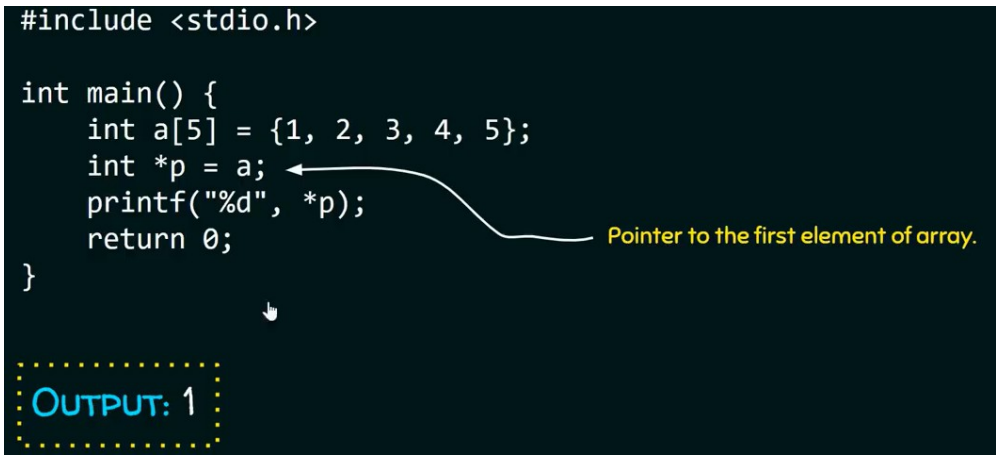
**a=>1000

***a=1;

Same logic for 3D as 2D array.

----------x----------

*Manually point to entire elements of an array.



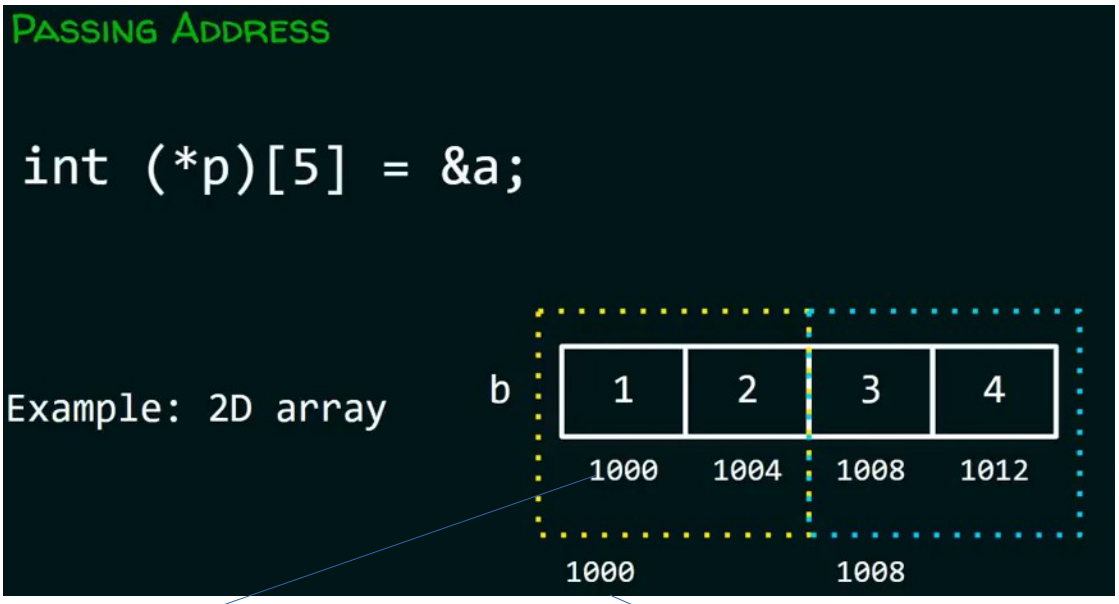
*Replace “int *p=a;” with*

*int (*p)[5]=&a; // means pointer to the whole array of 5 elements.*

Why using “&a” instead of “a”?

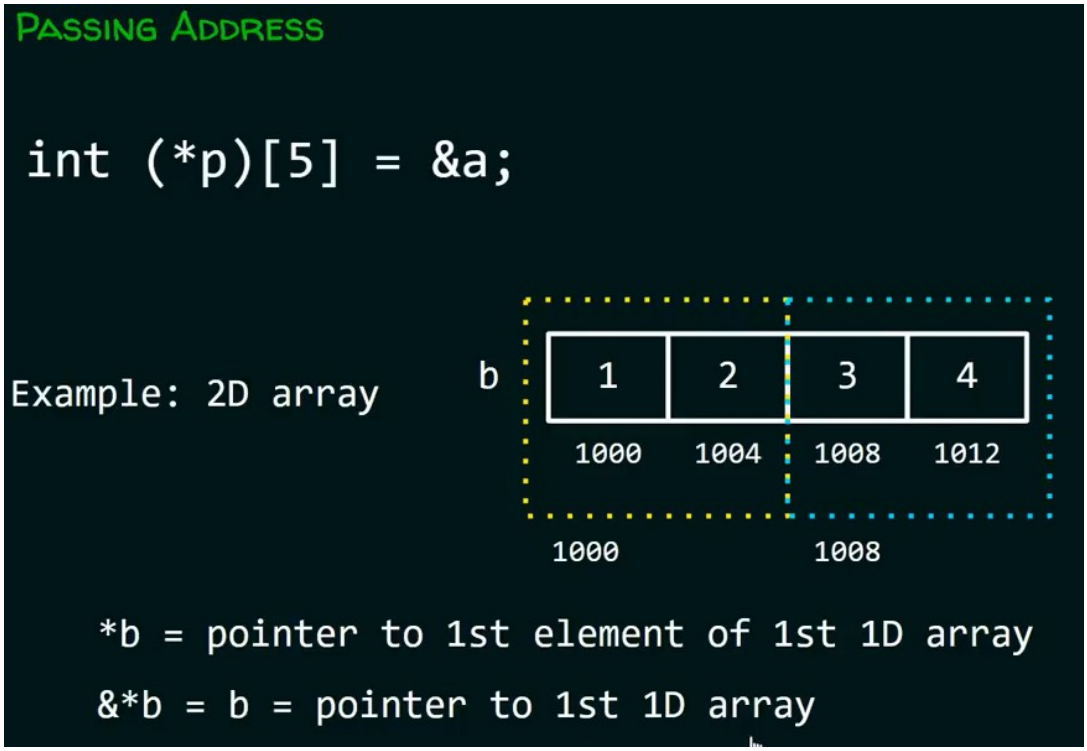
ans:

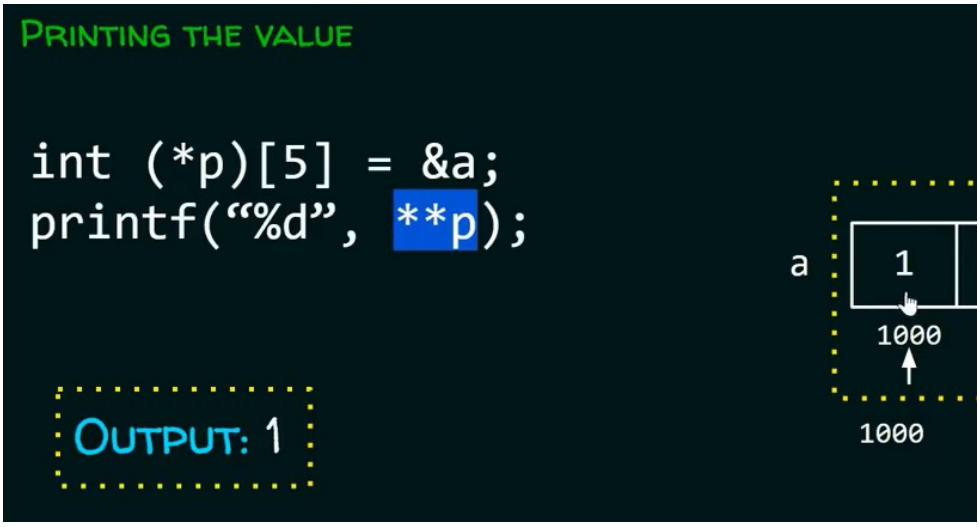
*Logic's are same as 2D and 3D array. "P" points to the whole array. When writing "*p" it get inside and points to the 1st element of the array.*



*So "*p" ==a; // as we know "a" points to 1st element of the array.*

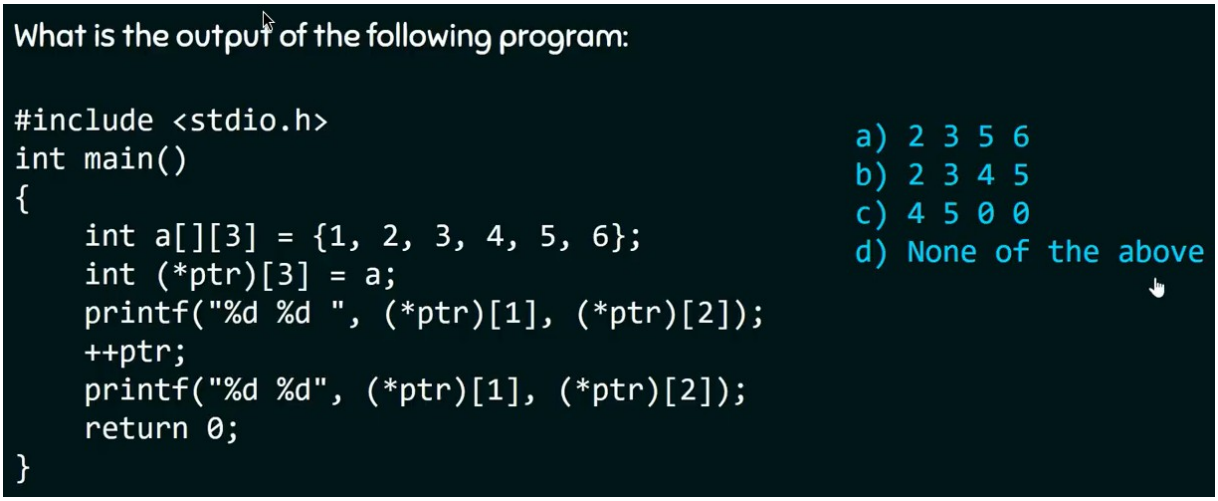
To come outside and points to whole array we need the "&" (as address(1000, outer address) of the address(1000, inner address) of 1st element)




PRINTING THE VALUE
int (*p)[5] = &a;
printf("%d", **p);
OUTPUT: 1
a
1
1000
1000

----------x---------

Example:


What is the output of the following program:
#include <stdio.h>
int main()
{
int a[][3] = {1, 2, 3, 4, 5, 6};
int (*ptr)[3] = a;
printf("%d %d ", (*ptr)[1], (*ptr)[2]);
++ptr;
printf("%d %d", (*ptr)[1], (*ptr)[2]);
return 0;
}
a) 2 3 5 6
b) 2 3 4 5
c) 4 5 0 0
d) None of the above

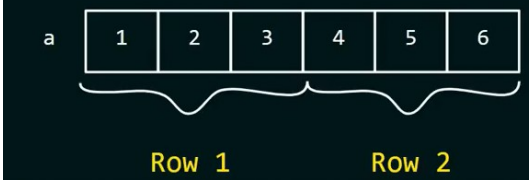

(*ptr)[1] = *((*ptr) + 1)

```c
#include <stdio.h>
int main()
{
    int a[][3] = {1, 2, 3, 4, 5, 6};
    int (*ptr)[3] = a;
    printf("%d %d ", (*ptr)[1], (*ptr)[2]);
    ++ptr;
    printf("%d %d", (*ptr)[1], (*ptr)[2]);
    return 0;
}
```

ptr is pointer to first 1D array

*ptr is pointer to first element of first 1D array

(*ptr)[1] = *((*ptr) + 1)

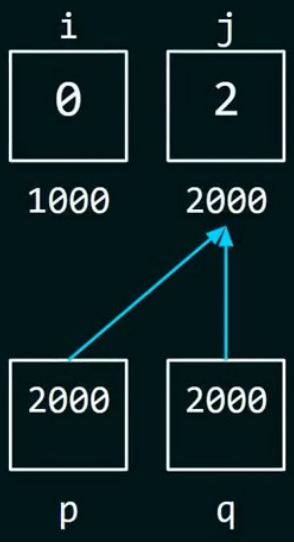


Ans: 2 3 5 6

---------x--------

What is the output of the following C program:

```c
void f(int *p, int *q)
{
    p = q;
    *p = 2;
}

int i = 0, j = 1;
int main() {
    f(&i, &j);
    printf("%d %d\n", i, j);
    return 0;
}
```



-------x-------

What is printed by the following C program:

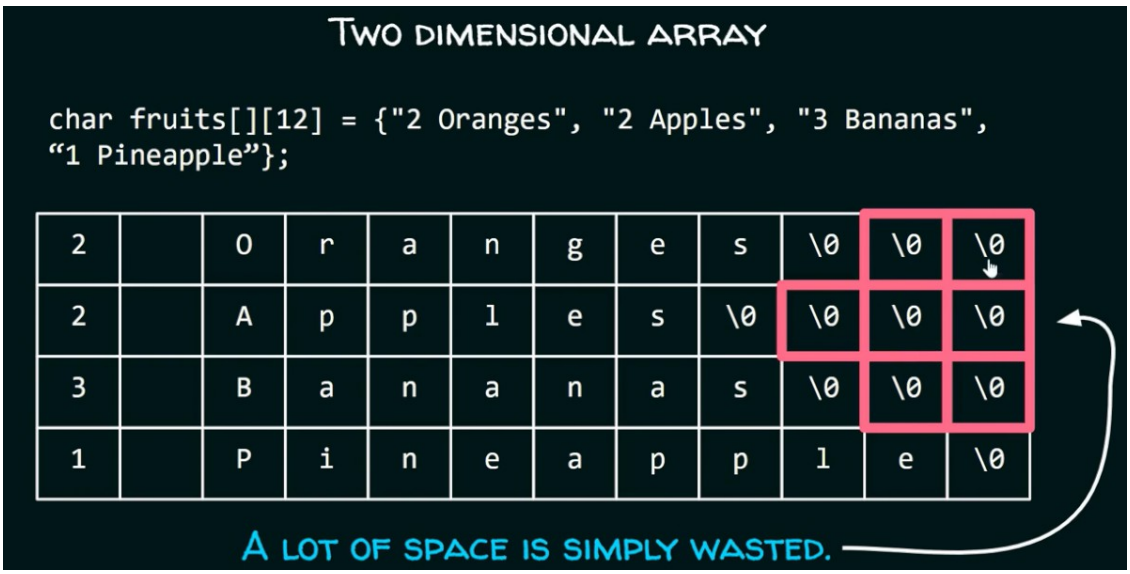
```
int f(int x, int *py, int **ppz)
{
  int y, z;
  **ppz += 1;
   z  = **ppz;
  *py += 2;
   y = *py;
   x += 3;
   return x + y + z;
}

void main()
{
   int c, *b, **a;
   c = 4, b = &c, a = &b;
   printf( "%d", f(c,b,a));
}
```

a) 18
b) 19
c) 21
d) 22

[GATE 20

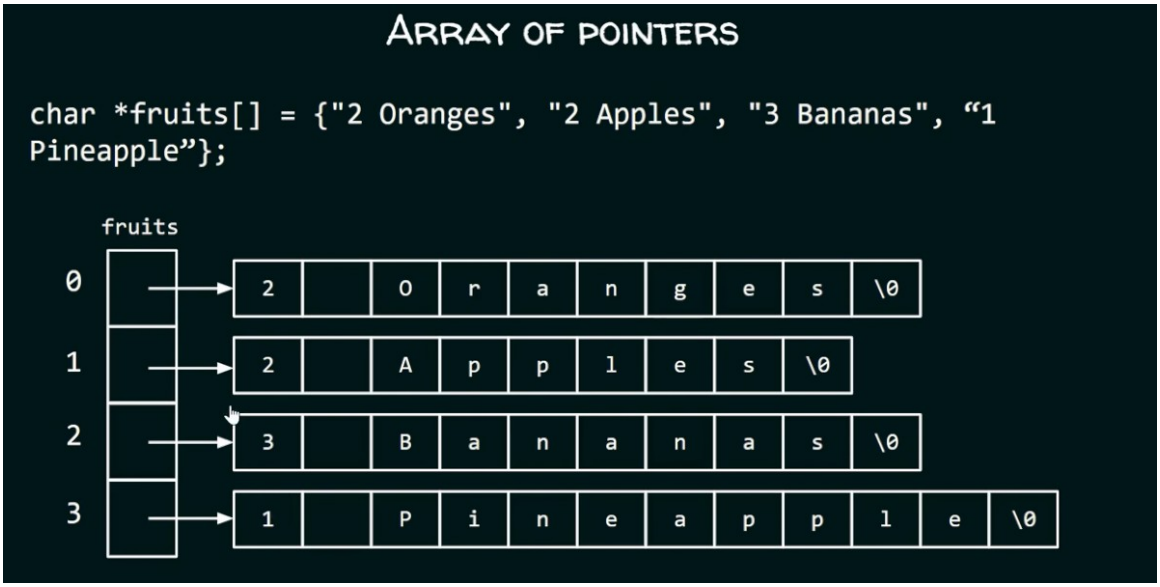
Ans: 19 // see lec-125 no lecture(pointer problem 7)

Array of pointers:

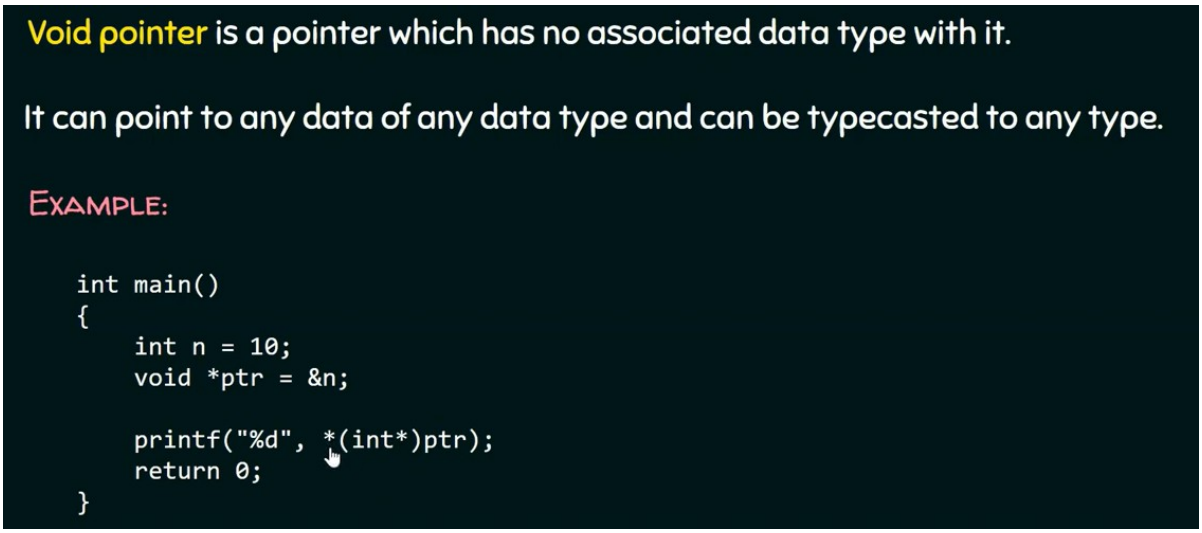


Wastage of memory cause of null character. Instead of it we use array of pointers.

See lec 142. Array of string



Void Pointer:



We can't dereference void pointer directly. We have to typecast to use it. See ***printf*** *part on above example.*

Use of void pointer(why we use): malloc and calloc function returns void pointer. Due to this reason, they can allocate memory for any data type.

NULL pointer: *Null pointer is pointer which does not point to any memory location. It represents an invalid memory location.*

```
int main() {
    int *ptr = NULL;
    return 0;
}
```

Uses:

** It is used to initialize a pointer when that pointer isn't assigned any valid memory address yet.*

** use when a memory is released.*

**** The value of NULL is 0. This '0' is not same as integer value '0'. We can use '0' or 'NULL'. Both same.*

*** Size of NULL pointer may depends on platform. Usually 8 byte.*

Dangling Pointer: *A pointer which points to non existing memory location. Suppose, a memory allocate with pointer. When that memory freed. Pointer which is pointing to that memory, is pointing to deallocated memory. This type of pointer called dangling pointer. To avoid this set pointer to NULL after freed memory.*

```
int main()
{
    int *ptr = (int *)malloc(sizeof(int));
    ...
    ...
    free(ptr);
    ptr = NULL;
    return 0;
}
```

Now, ptr is no more dangling.

Wild Pointer: *Wild pointers are also known as uninitialized pointers.*

Wild pointers are also known as uninitialized pointers.

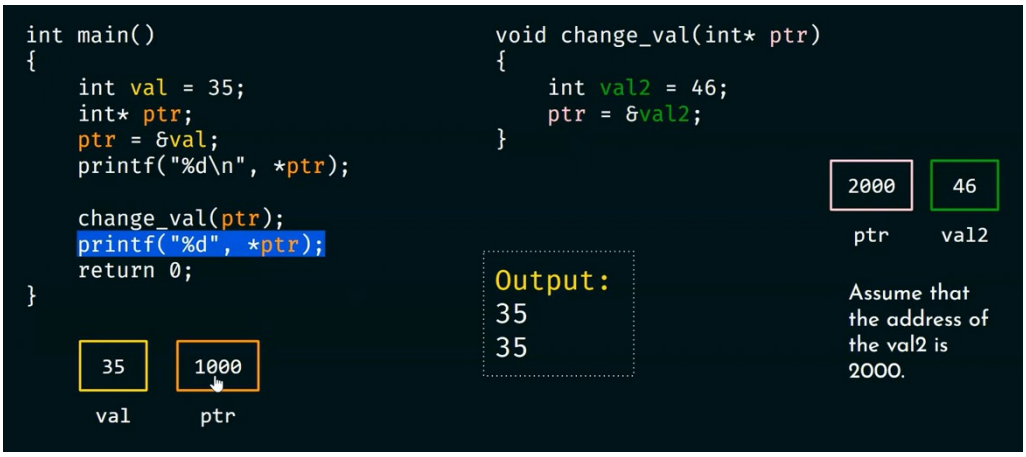
These pointers usually point to some arbitrary memory location and may cause a program to crash or misbehave.

EXAMPLE:

```
int main()
{
    int *p;
    *p = 10;
    return 0;
}
```

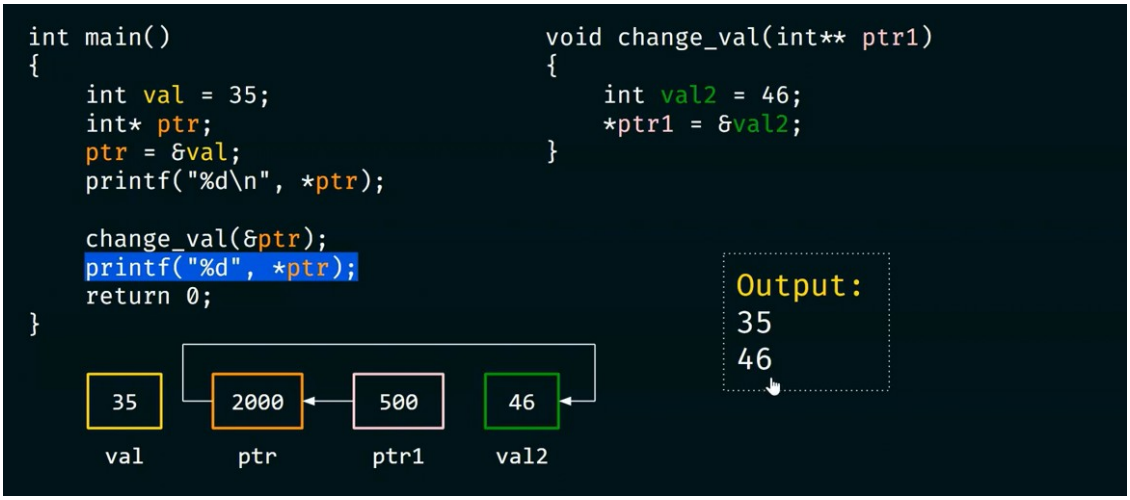
Here pointer P is pointing to any arbitrary memory. That memory may occupy another program. Accessing that memory may cause crash.

Pointer to Pointer(Double Pointer):



Here above a copy of a pointer is passing(pass by value). That's why value not changing.

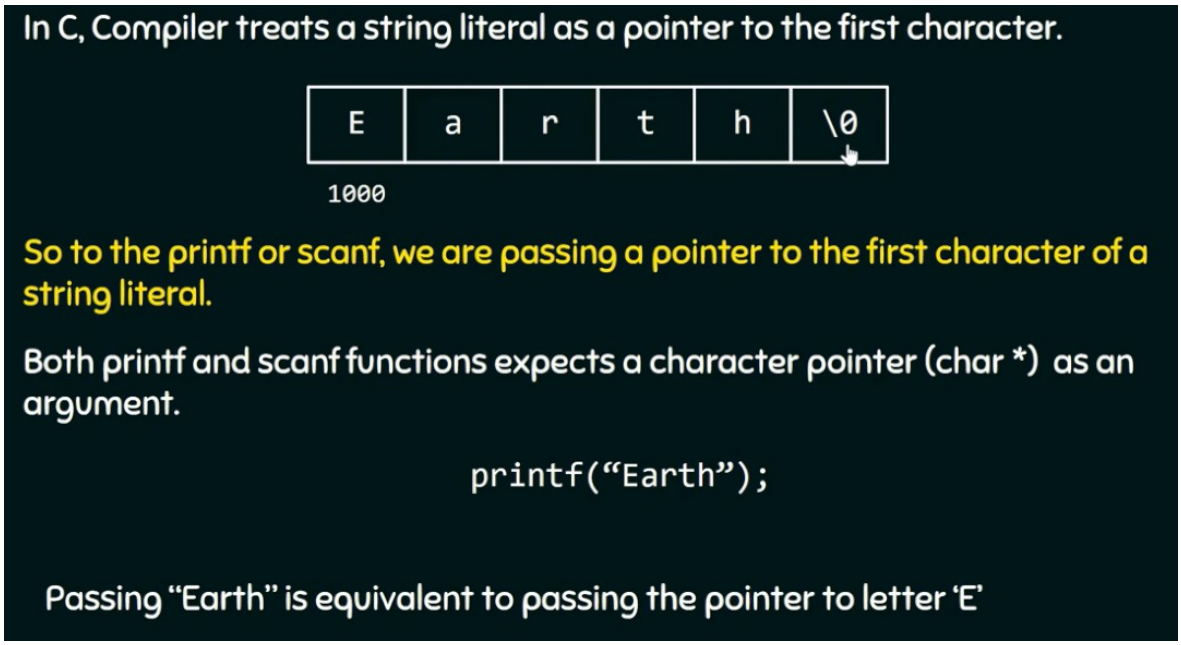
If we pass reference

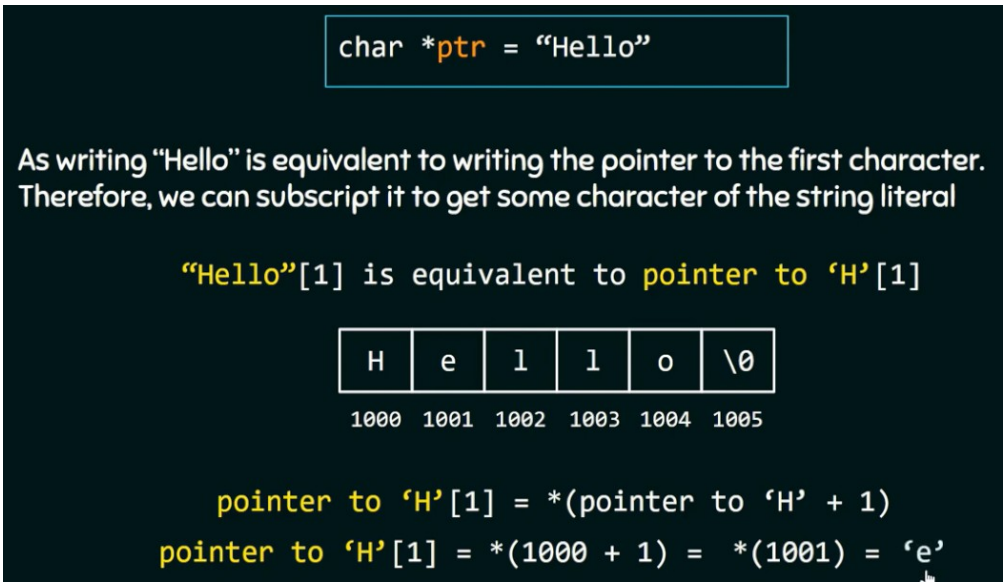


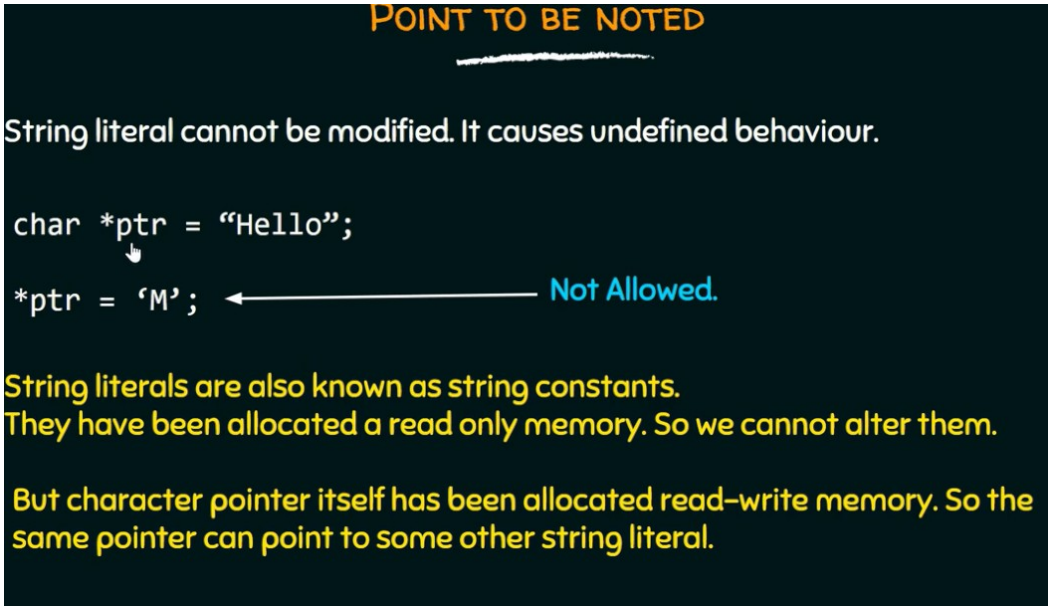
----------x--------

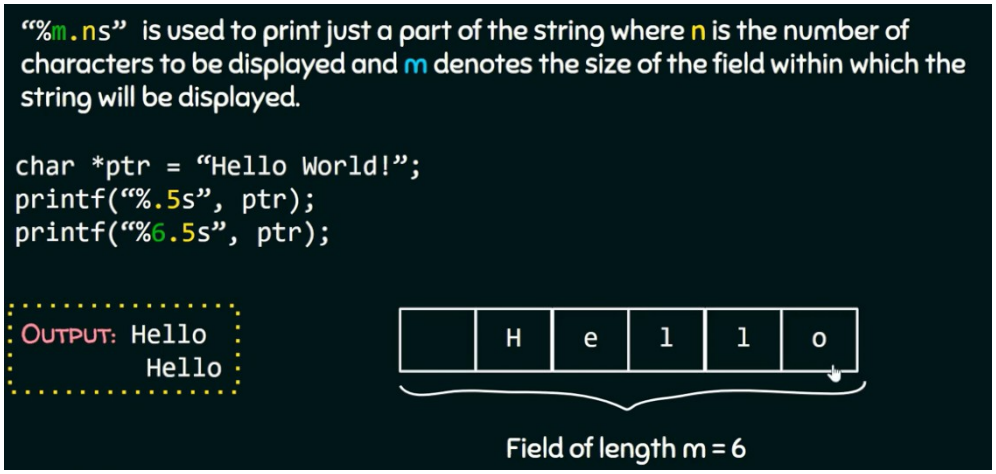
String:

**String literals are stored as array of characters. It is always ended with '\0'(null character). which indicates the end of the string.*









Puts(): *puts() function used to write string to the output string. This automatically add a new line after string.*

Scanf():

scanf("%s", a);

** here 'a' is treated as a pointer of the first element of the array. So no need to put '&' like int variable.*

** it store all character except white space. After white space it ignore all character. "you are fool". Here scanf will store 'you'*

gets(): *This is used to read entire line as input.*

**** scanf() & gets() both function have no way to detect when the character array is full. May cause undefined behavior, may lead to buffer overflow error. Program may crash.*

Although, scanf() has the way to set the limit for the number of characters to be stored in the character array.

By using %ns, where n indicates the number of characters allowed to store in the character array.

```
char a[10];
printf("Enter the string:\n");
scanf("%9s", a);
printf("%s", a);
```

But, unfortunately, gets() is still UNSAFE.

It will try to write the characters beyond the memory allocated to the character array which is unsafe because it will simply overwrite the memory beyond the memory allocated to the character array.

getchar(): *this function is used to read 1 character at a time from the user input. It returns integer value of that inputed character ASCII value.*

Putchar(): *putchar() take int value as argument, and print character of the corresponding int.*

int a=65;

putchar(a); /// output>> "A"

****strcpy():*** *strcpy defined on <string.h> header file. It returns the pointer of the 1st character of copied string.*

Prototype: `char* strcpy(char* destination, const char* source)`

It isn't modified. That is why it is constant.

strcpy is used to copy a string pointed by source (including NULL character) to the destination (character array)

Example:

```
#include <stdio.h>
#include <string.h>

int main() {
    char str1[10] = "Hello";
    char str2[10];

    printf("%s\n", strcpy(str2, str1));
    printf("%s", str2);
    return 0;
}
```

strcpy returns the pointer to the first character of the string which is copied in the destination.
Hence if we use %s, then whole string will be printed on the screen.

Output: Hello

output: both will print "Hello"

Caution: strcpy(str2, str1);

Here str2 will copy to str1. If str2 size is greater than str1, then there will be undefined behavior. No way to check whether str2 will fit to str1. I that case we should use ***strncpy().***

Strncpy():

```
strncpy(destination, source, sizeof(destination));

Example:

#include <stdio.h>
#include <string.h>

int main() {
    char str1[6] = "Hello";
    char str2[4];
    strncpy(str2, str1, sizeof(str2));
    printf("%s", str2);
    return 0;
}
```

OUTPUT:
Hell

Here only character will copy according to size of destination(str1).

*** ***strncpy*** *never add null character(\0) in the end. It is necessary to add.*

str2[sizeof(str2)-1]='\0';

strncpy will leave the string in str2 (destination) without a terminating NULL character, If the size of str1 (source) is equal to or greater than the size of str2 (destination).

strlen(): *it returns the size of string.*

** it doesn't count null character.*

```
#include <stdio.h>
#include <string.h>

int main() {
    char str[100] = "Hello World";
    printf("%d", strlen(str));
    return 0;
}
```

OUTPUT: 11

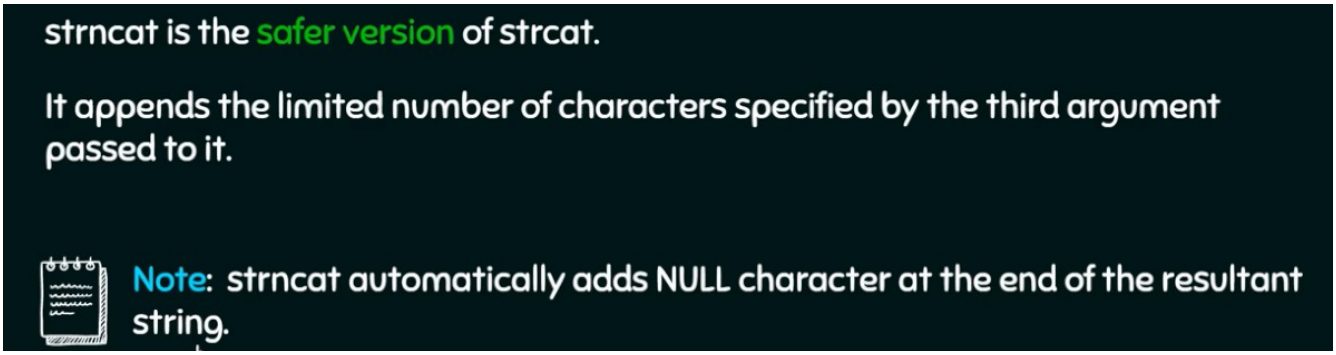
It calculates the length of the string and not the length of the array.

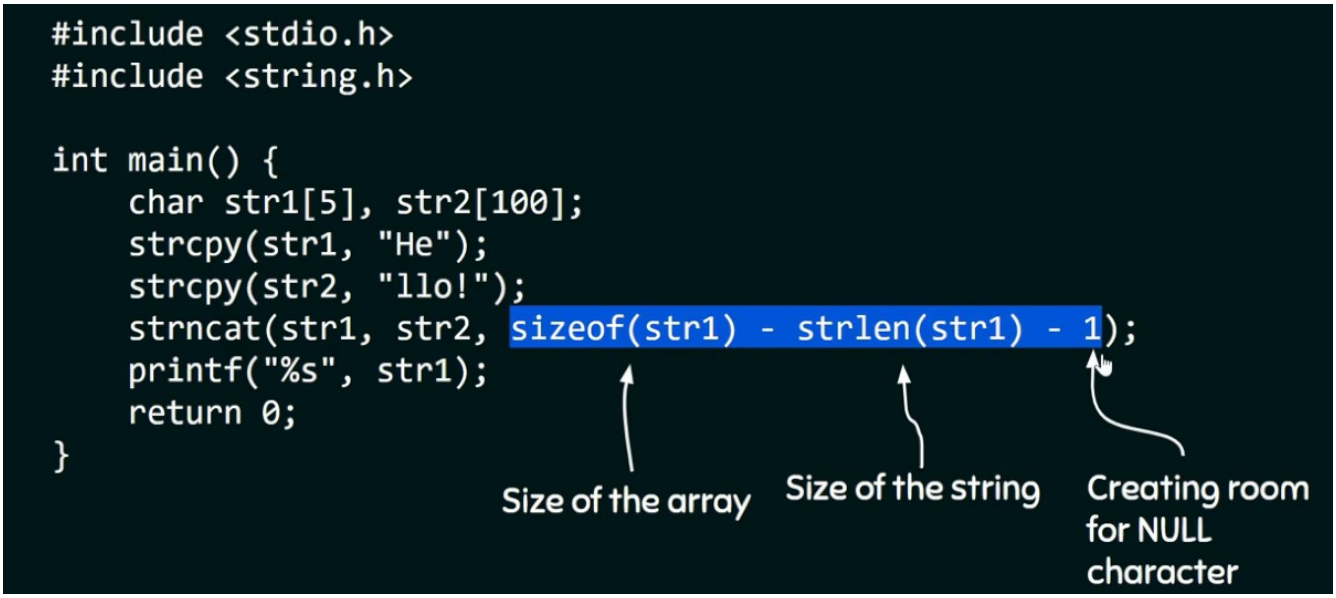
strcat(): *string concatenation function.*

```c
char str1[100]="Hello";
char str2[100]=" World";
strcat(str1, str2);
printf("%s\n", str1);
```

Here str2 will add at the end of str1.

*** str1 size have to enough to concatenate str2. Otherwise undefined behavior may occur. So we can use* ***strncat.***

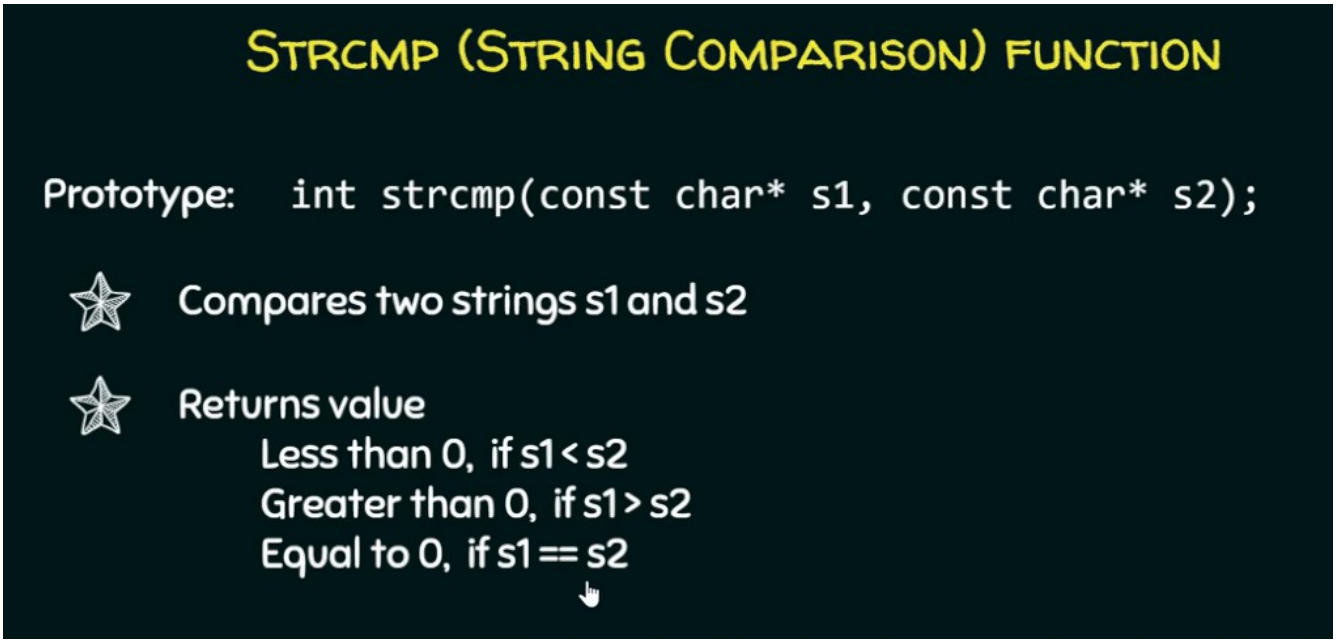


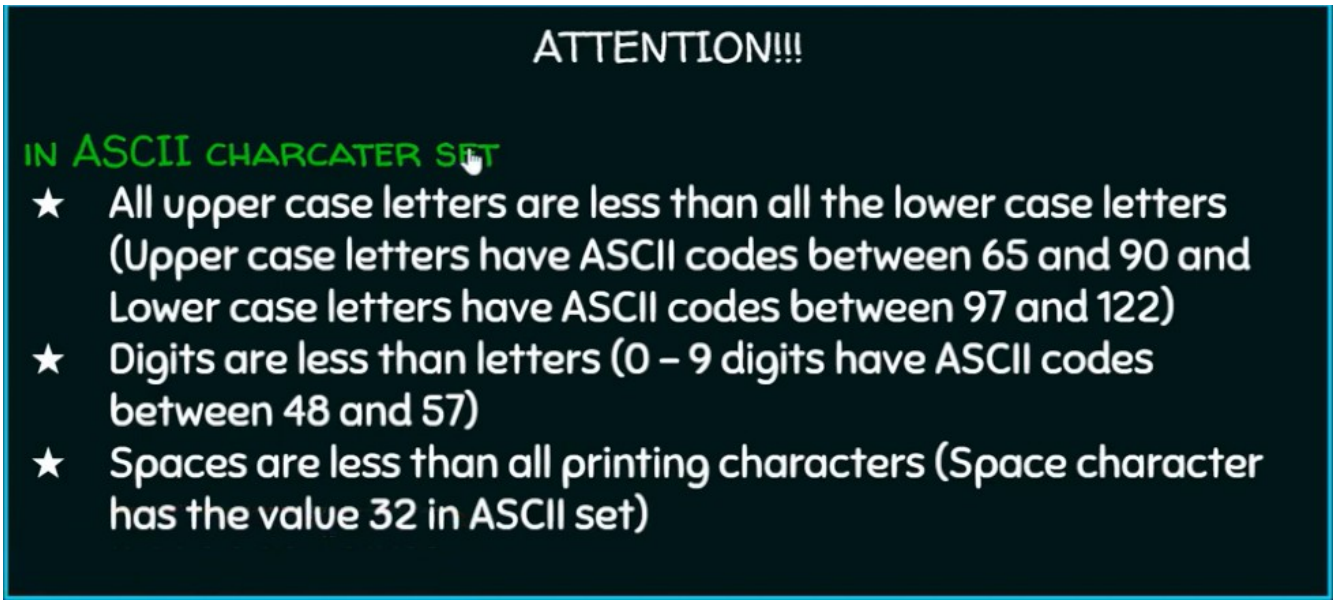


Ans: "Hell" *// 5-2-1=2 onlyy 2 character will concatenate*

** strncat add '\0' at the end of concatenating string.*

strcmp():





See Lec 143-147 (String problems)

Function Pointers: *Function pointers are like normal pointers but they have capability to point to a function. Function are set of instructions which are stored in a memory. And they have memory location. Pointer point to that function location.*

*** ***int *ptr[10];*** *//10 pointer pointing to array elements.* ***See lec 148 (function pointers in C)***

so we ca write:

int (*ptr)[10]; *// **ptr** is a pointer which pointing to an array(of 1st element), first brackets used caused '[]' precedence is higher than '*'.*

Declaring a function:

int (*ptr)(int, int)=&add; *//here '**add**' is a function not written here. It means **ptr** is a pointer which point to function which have 2 arguments. And returns an integer. After assigning '&add' it will point to that function.*

Or

int (*ptr)(int, int);

ptr=&add;

2 way we can write a function and call function with pointers.

```
int add(int a, int b)
{
    return a+b;
}

int main()
{
    int result;
    int (*ptr)(int, int) = &add;
    result = *ptr(10, 20);
    printf("%d", result);
}
```

```
int add(int a, int b)
{
    return a+b;
}

int main()
{
    int result;
    int (*ptr)(int, int) = add;
    result = ptr(10, 20);
    printf("%d", result);
}
```

differences: *if we use '&' we need to put '*' as dereference operator. Otherwise don't need both.*

Replace: *result=(*ptr)(10,20);*

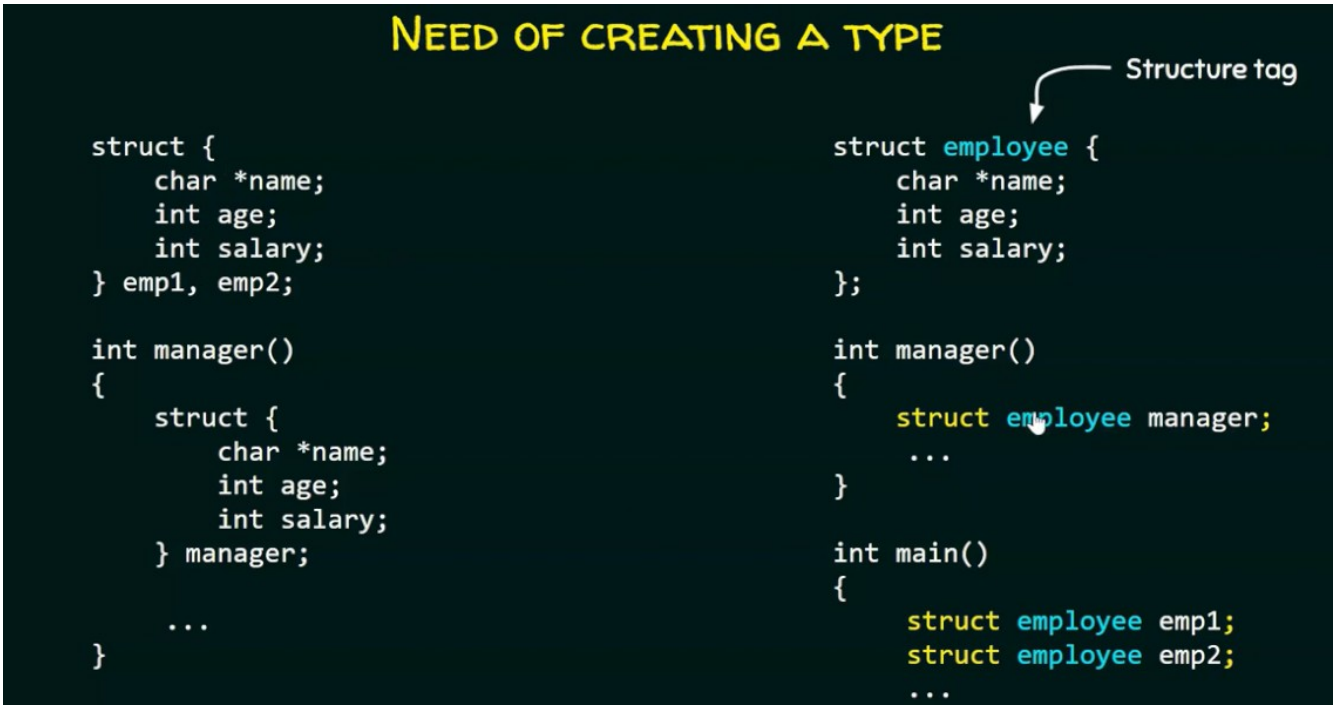
Application of function pointers:

calculator: Where we can use if/switch case. But we can use function pointers. ***See lec 149: (application of function pointer)***

---------x--------

Structure: *Structure is a user defined data type that can be used to group elements of different types into a single type.*

Structure Tag: *Structure tag is used to identify a particular type of structure.*



Typedef: Typedef gives freedom to the user by allowing them to create their own type.

Syntax: tyfedef existing_data_type new_data_type

```
struct car {
    char engine[50];
    char fuel_type[10];
    int fuel_tank_cap;
    int seating_cap;
    float city_mileage;
};

int main() {
    struct car c1;
}
```

We can write this with typedef… mentioned below


```
typedef struct car {
    char engine[50];
    char fuel_type[10];
    int fuel_tank_cap;
    int seating_cap;
    float city_mileage;
}car;

int main() {
    car c1;
}
```

car becomes a new data type.

Structure Padding:

When an object of some structure type is declared then some contiguous block of memory will be allocated to structure members.

For example:

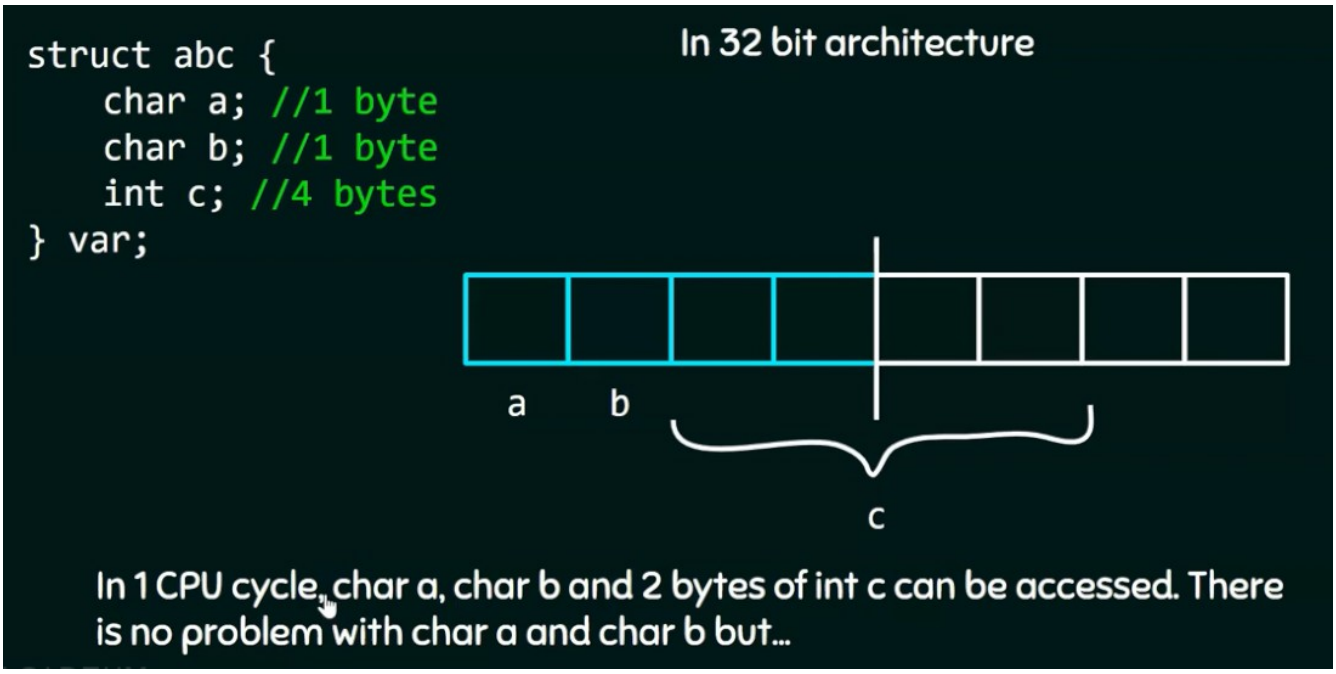
```
struct abc {
    char a;
    char b;
    int c;
} var;
```

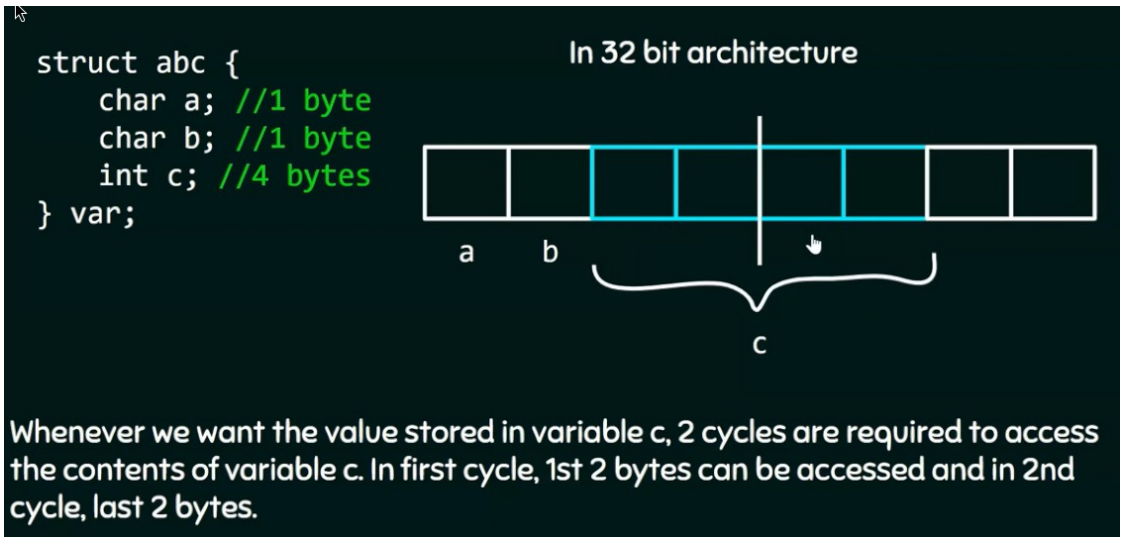
Here 'var' is an object. Here we can assume structure 'abc' should have 6 byte. But wrong.

There is a concept 'Structure padding'

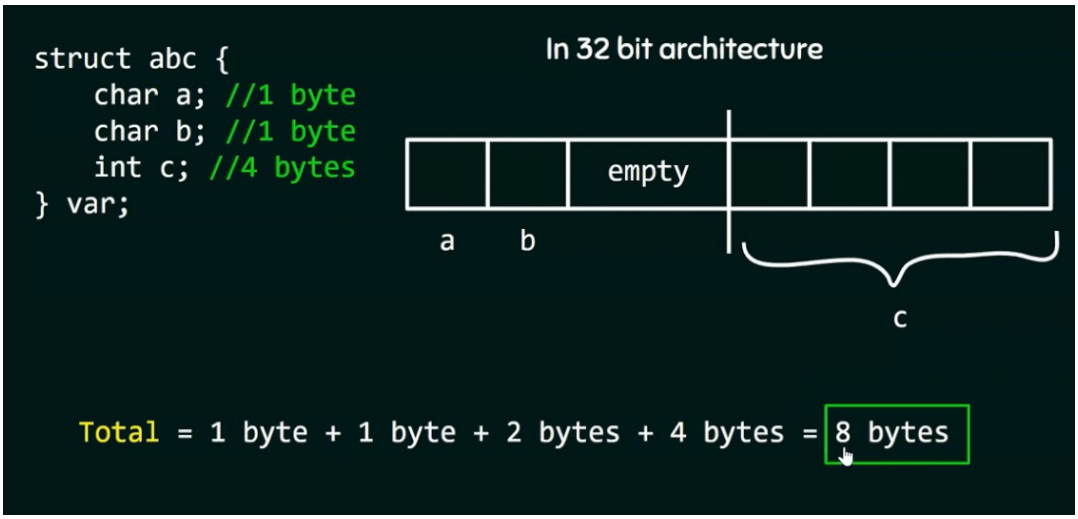
Processor doesn't read 1 byte at a time from memory. It read 1 word at a time.

This means: *If we have 32 bit processor, it means it can access 4 byte(32 bit) at a time, which means word size is 4 byte. For 64 bit processor word size is 8 byte.*

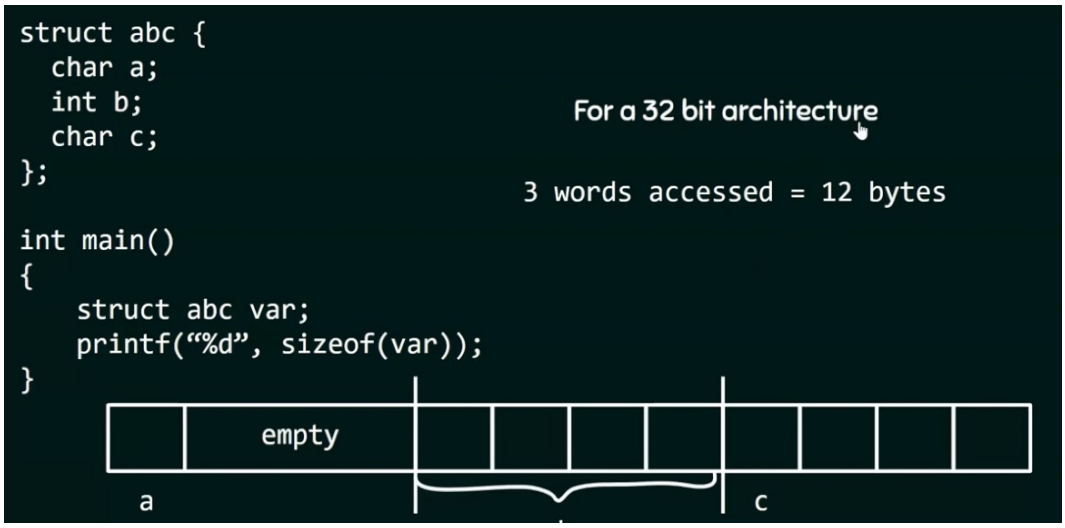




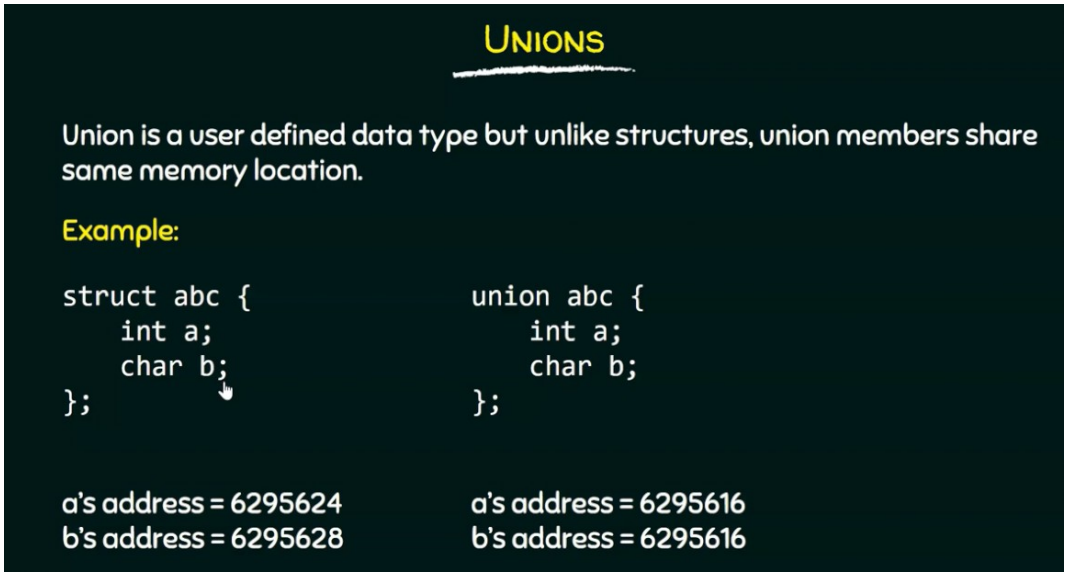
Accessing int c, required 2 cpu cycle. Wastage of cpu cycle. Using padding concept, empty room will be created here.



Order of member my effect the size of an object…



Union:



Size of the union is taken by size of the largest member of the union.

See size diff bet structure & union

```
typedef union {
    int a;
    char b;
    double c;
} data;                          Size = 8 bytes

int main()
{
    data arr[10];   Size = 80 bytes
    arr[0].a = 10;
    arr[1].b = 'a';
    arr[2].c = 10.178;
    //and so on
    return 0;
}
```

```
typedef struct {
    int a;
    char b;
    double c;
} data;                          Size = 13 bytes

int main()
{
    data arr[10]; Size = 130 bytes
    arr[0].a = 10;
    arr[1].b = 'a';
    arr[2].c = 10.178;
    //and so on
    return 0;
}
```