C PROGRAMMING COURSE BY GO CLASSES

Lectures by Sachin Mittal

References :

- The C programming Language by Dennis Ritchie

1. INTRODUCTION AND NUMBER SYSTEM

Why we need to study C programming ? – We need to compute in order to perform task (basically we want to communicate with computer)

1.1) Introduction :

What we are going to see in this lecture notes ? –

1) Unsigned and signed integers
2) Data types
3) Control statements
4) Operator precedence
5) Functions
6) Memory layout and storage classes
7) Recursion
8) Pointer
9) Arrays (1D, 2D, 3D, passing arrays to function)
10) Memory leak and dangling pointer
11) Char array vs strings
12) Pointer to array vs array of pointers
13) Structs and Union
14) Complex declarations
15) Miscellaneous

Let's look at one c program,

```
#include <stdio.h>
int main(void){
    printf("Hello world!\n");
}
```

Here we are not worry about meaning or specification of these keywords, we are only interested in working or output of program.

1.1.1) Data types and their properties in C programming :

1) int (4 bytes)
   - signed
   - unsigned
2) Char (1 byte)
   - signed
   - unsigned
3) Short (2 bytes)
   - signed
   - unsigned
4) float (4 bytes)
5) double (8 byte)
6) long double (10 bytes)

What will happen if you write int x = 5; first this is int so 4 bytes will be allocated to variable x then it will store 5 in binary number at some location in memory.

Formate specifier in C programming :

Char (%c)

Int (%d)

Unsigned int (%u)

Float (%f)

Hexadecimal format (%x) – 0x & octal - x

Addresses (%p)

Strings (%s)

Double (%lf)

This is not all formate specifier, we will look overtime about more of these format specifiers.

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

we have to specify formate while printing values.

Output : 5

1.1.2) Detour – Number system :

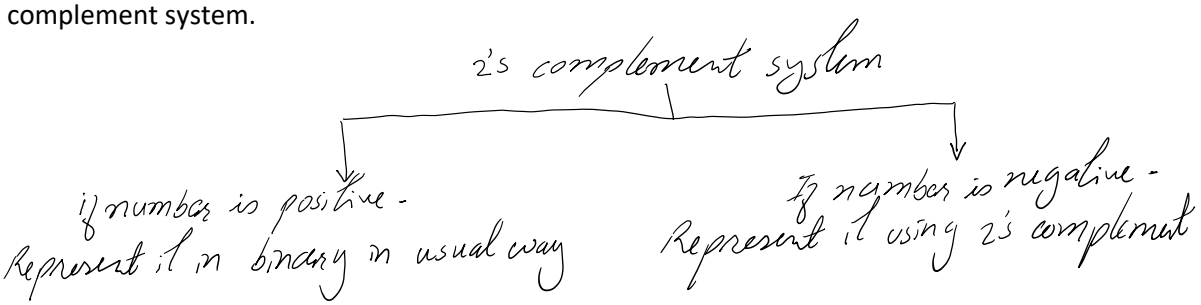
Now we will see how variables (positive or negative) are stored in memory. Let's say

```
int a = -5; int a = 3;
```

For this we have to see how computer converts decimal to binary number.

1) **Decimal to binary** : We keep dividing by 2 and save remainder this is one method and second is to use 1+2+4 formate (this is time saving).
2) **Binary to decimal** : You know this no need to explain.

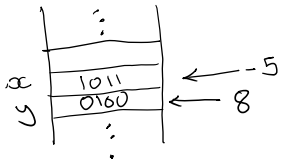
But how to represents negative numbers or integers ? – for this we have to understand 2's complement system.



In 2's complement system, MSB will indicate the sign of the number : 0 if positive and 1 if negative.

How to compute 2's complement ? – Take 1's complement of number and add 1. **The 2's complement of number is obtained by leaving the all least significant 0's and the first 1 unchanged and then replacing 1's with 0's and 0's with 1's (Complement).** Example : 8 = 0100 (positive since MSB is 0) = 1100 (after taking 2's complement) but as it is positive computer does not store it in 2's complement, it will store it in usual binary form. But let's say x = -5 = -(0101) = 1011 (in 2's complement)

In memory,



But why we considered only 4 bits why not 3 bits (what's wrong with representing 5 as 101) ? – We cannot represent 5 using 3 bits in 2's complement number system.

The number 43 in 2's complement representation is

a. 01010101

b. 11010101

c. 00101011

d. 10101011

Answer : 43 in binary is 00101011 take 2's complement you will get A as answer but this is wrong. Number is positive so we do not have to take 2's complement. We do 2's complement of number which are negative this is already positive so answer is C.

3) 2's complement to decimal :

- ***Method 1*** **:**

$$0101 = -0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 0 + 4 + 0 + 1 = 5$$

$$1011 = -1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = -8 + 0 + 2 + 1 = -5$$

Consider another example, for -28 we first convert 28 to binary and then to 2's complement because we want negative value of 28. So, 28 = 11100 and its 2's complement is …11100100.

- ***Method 2*** **:**

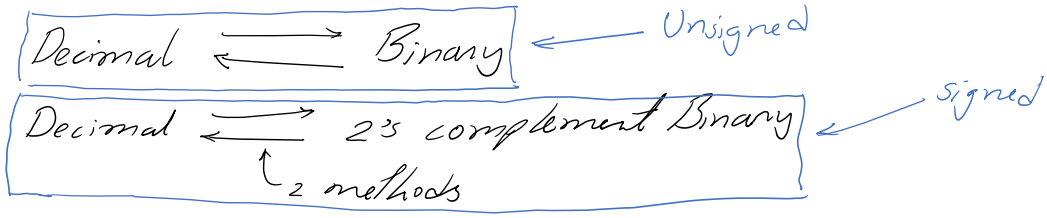
If the number is positive (the most significant bit is 0), convert the number into decimal formate normally.

If the number is negative – Take 2's complement of number and then convert to decimal. Put the minus sign.

Example, take 1101 now in signed representation it is negative as 1 is there at MSB. In this case we simply take 2's complement and put the minus sign. So, in this case 1101 = 0011 after doing 2's complement, which is 3 in binary and then we put minus sign to get -3 as answer. You should follow this method 2.

Given a representation in bit form 1010, what is the corresponding value ? – before converting this number to decimal first we have to ask one question that is this conversion is from signed to decimal or unsigned to decimal. Nothing is given so we convert it in both. In unsigned it is 10. And in signed it is by method 2, -6.

So far what you know,



4) Copying number to higher bit representation :

Why we want this because in c programming we have operation like,

```
short int x = 4;
int y = x;
```

In this example, we are converting short int (16 bits) which has lower bits than int (32 bits). So, how compiler convert this number that we have to see.

- ***Copying Unsigned number (also known as zero extension)*** :

Consider 4 bit unsigned 0101, we want to copy it to 8 bits. We copy 0101 as it is and then we add zeros in front of that number. So, in 8 bits we will get 0000 0101.

- ***Copying signed number (also known as sign extension)*** :

Consider 8 bits representation of some integer : 1001 0110. And we want to convert it into 9 bits representation. So, should we write 0 at the front or 1. Let's see. If we put 0 at the front then clearly 8 bits representation will not be same as 1001 0110 in signed is negative and if we put 0, 0 1001 0110. This is positive so idea of putting 0 did not work. Now let's put 1 instead. Yes, it is true. And one beautiful thing to notice is that.

1001 0110 = 1 1001 0110 = …1 1001 0110

In copying signed number rule is we need to copy sign bit (0 or 1) as it is

Example, 8 bits representation of some integer : 1001 0110

9 bits representation of the same integer : 1 1001 0110

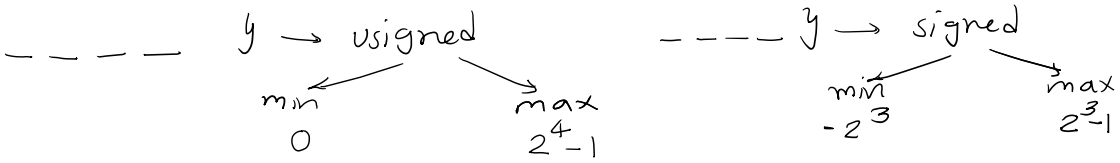
11 bits representation of the same integer : 111 1001 0110

8 bits representation of some integer : 0101 0110

9-bits representation of the same integer : 0 0101 0110

Now, we have very good understanding of signed and unsigned number so, let's address few questions,

Min and max number in unsigned and signed form with k bits. – let's first find with 4 bits and then we extend out idea to k bits.



So, in unsigned form with k bits : 0 to 2^k-1

In Signed form with k bits : -2^{k-1} to 2^{k-1}-1

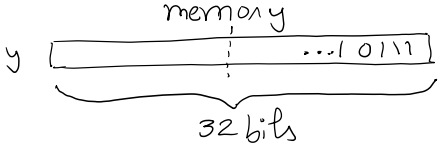
1.2) Numbers in C programming :

Now, we have seen how signed and unsigned numbers are represented in c programming we will see how those numbers are stored in memory and more.

1.2.1) Signed and unsigned numbers in memory :

```
int y = -9;
printf("%d", y);
```

← 32 bit



```
printf("%u", y);
```

This says extract the number from y variable and execute then as signed number or whatever it is given (signed or unsigned). *Output* : -9

This says extract the number from y variable as unsigned number (%u is formate specifier of unsigned number). *Output* : very large number as first all bits upto 32 are 1.

Now, consider we put unsigned in front of int y = -9. Then ? – forget about unsigned or int, -9 will be stored as signed then if you printf("%d" then as I previously said it extract the number from y variable and execute whatever it is given (signed or unsigned). Here -9 is signed although it is written as unsigned in front of int y. then we have %u which acts same as previously resulting in very large number.

printf doesn't use the information about type of variable

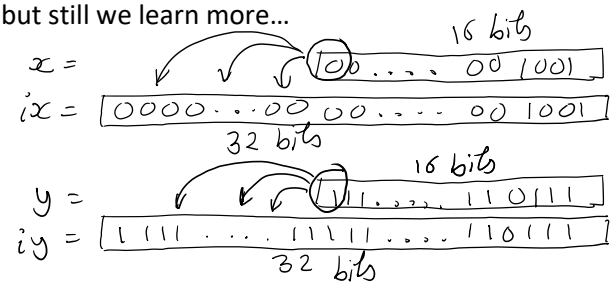
1.2.2) Extension and truncation :

Extension (zero or sign) : Copying a lower bit number to higher bit. Eg – short to int

Truncation : copying a higher bit number to lower bit. Eg – int to short

1) **Extension** : We have already seen this but still we learn more…

```
short int x = 9;
int ix = x;
short int y = -9;
int iy = y;
```

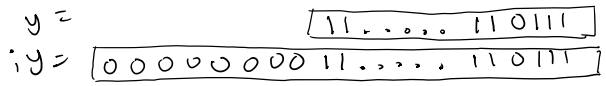


while copying from x to ix we don't care about the type of ix. Whatever is x (signed or unsigned) we store in ix. Let's take another example :

```
short int y = -9;
unsigned int iy = y;
```

Here iy is unsigned so we should store y in unsigned formate but in c programming extension, we do not care about iy we simply put y in iy with signed or unsigned is depend upon the data type of y.

```
unsigned short int y = -9;
int iy = y;
```



Here we do not copy 1 as source (y) is unsigned we extent it with zero. In previous example y was signed so we copied 1.

Extension depends on RHS (source)

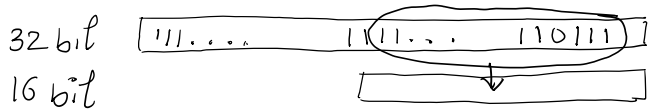
- Promotion always happens according to the source variable's type
  Signed : "sign extension" (copy MSB – 0 or 1 to fill new space)
  Unsigned : "zero fill" (copy 0's to fill new space)

2) **Truncation (higher bits to lower bits)** :

- Regardless of source or destination signed/unsigned type, truncation always just truncates

- This can cause the number to change drastically in sign and value.

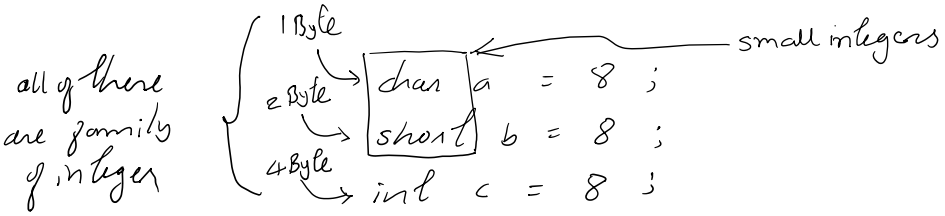
```
int y = -9;
short int iy = y;
```



1.2.3) Integer promotion :

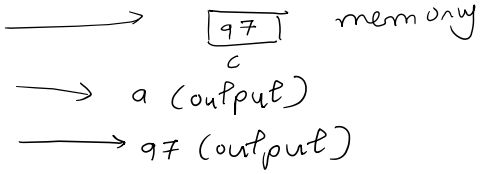
Whenever a small integer type (char or short) is used in an expression, it is implicitly converted to int.

Before explaining above sentence let's look at some basic facts,



If someone asks you that what will be the best way to store this 8 using minimum memory so answer is using char data type. Here one thing to note that char data type stores numbers in memory. Each character has some ASCII values associated with it.

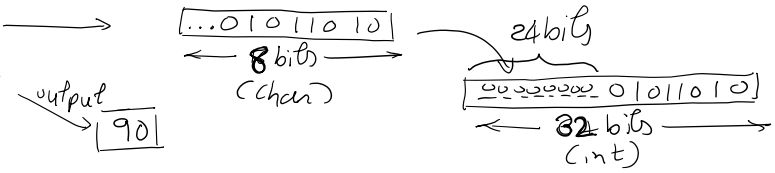
```
char c = 'a';
printf("%c",c);
printf("%d",c);
```



Now, let's understand the meaning of first sentence, It says whenever a small integer type (*char* or *short*) is used in an expression (like c^2-1 or c+0 or whatever) it is implicitly converted to *int* (c^2-1 or c+0 will become *int* type from *char* or *short*).

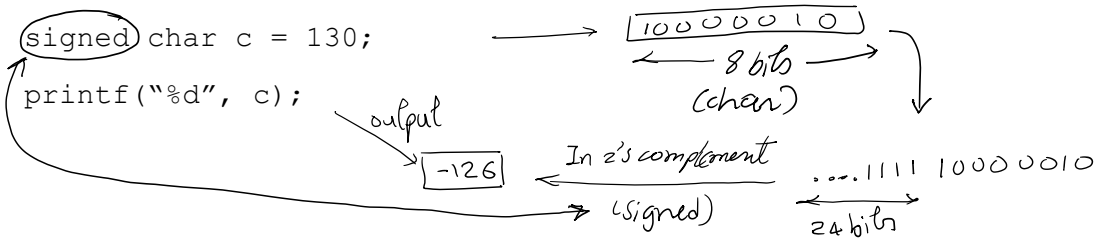
Now, let's see what's actually happening inside the memory,

```
char c = 90;
printf("%d", c);
```



Here while printing c it is first converted to int and then it is being printed this is called *integer promotion.* Now, consider following code,

```
signed char c = 130;
printf("%d", c);
```

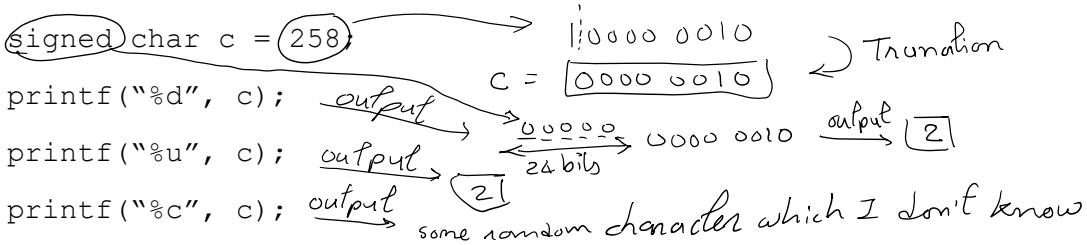


In above example, let's write %u (unsigned) instead of %d. It will result in huge number as 1111… all will be considered as unsigned big value.

What if you use unsigned char c instead of signed char c.

```
unsigned char c = 130;
printf("%u", c);
```

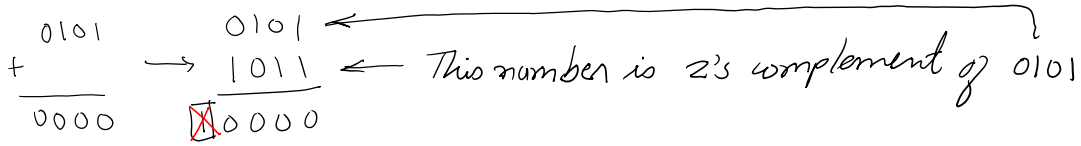
Now, what will happen if you store number to char which is larger than 1 byte.



Here we use concept of truncation.

1.2.4) Something strange about 2's complement :

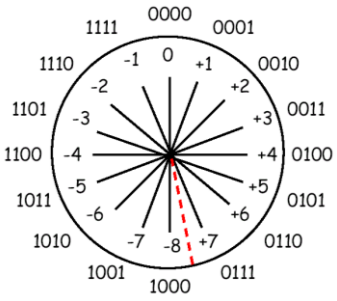
Consider you have game where you have to add something to given number to get all zeros. There may or may not be overflow so we ignore overflow.



Why this pattern ?

Very simple, 2's complement = 1's complement + 1

Any number + 1's complement of that number = all ones. And now you add 1 to get 2's complement you obviously get all zero ignoring overflow.



4-bit 2's complement signed integer representation

1.2.5) Type conversion in C programming :

Consider the following code and predict the output,

```
Int x = 10;
Int y = 3;
float res;
res = x/y;
printf("%f", res);
output : 3
```

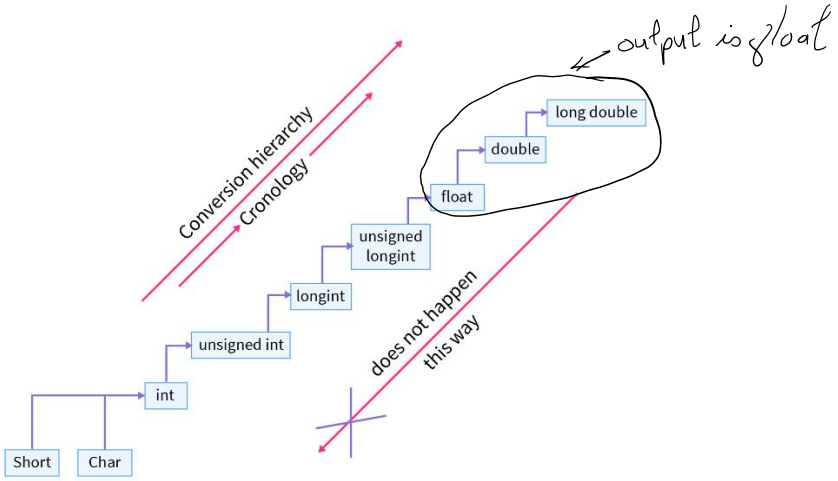
The result of this calculation is int

```
Int x = 10;
float y = 3;
float res;
res = x/y;
printf("%f", res);
output : 3.33...
```

Result of this operation is float

```
Int x = 10;
Int y = 3;
float res;
res = (float) x/y;
printf("%f", res);
output : 3.33...
```

Here output : 3.000



This even applies to signed/unsigned numbers in c programming language, if signed(i.e. lower) and unsigned(i.e. higher) is present under any expression (this also involves expression involving +, -, /, *) then it is first converted into unsigned(i.e. higher in general) and then calculation happens due to implicit type conversion.

C language allows implicit type conversion from a floating-point type (like double or float) to an integer type (int). so, if we write int a = 10.1; 10.1 is implicitly converted to int and thus no error.

1.3) Conditional statements in c programming :

The if statement, the switch statement,

1.3.1) If-else statements :

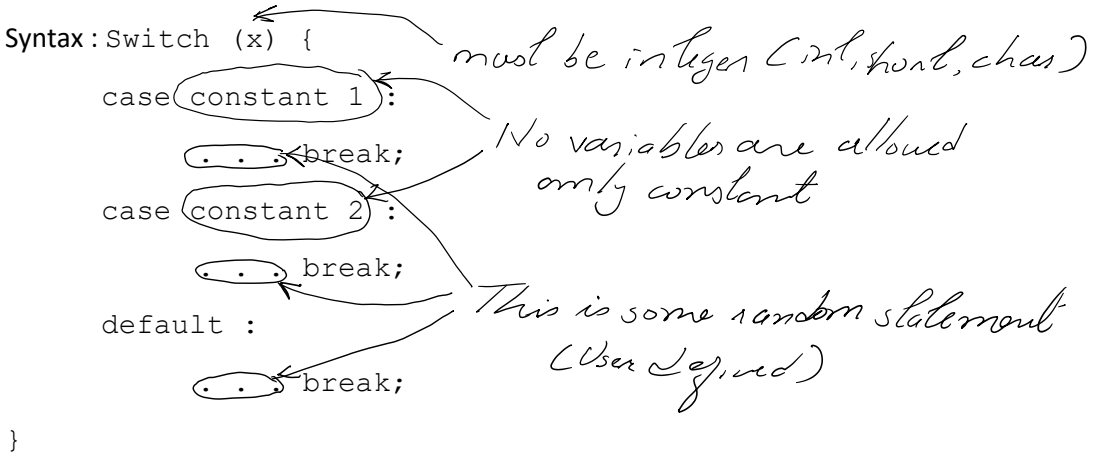
Syntax :
```
if (test expression) {
        Statement-block;
}
```
0 ⇒ False
otherwise ⇒ True

Now, we have problem with nested if-else. Consider following code and predict the output,

```
a = 5;
if(a<10)
    if(a % 2 == 0) printf("a is even and less than 10");
else printf("mystery");
```

Always remember that else belongs to closest if, if it is nested if-else unless it is not specified by brackets. So, output is mystery.

1.3.2) Switch-case statements :



The part of the case clause must be a constant integral value or a constant expression followed by a **colon** (not semi colon or something). It cannot contain any variables, unlike in the switch clause. Moreover, they should only be an integral/character constant (for example, case 'a' : or case 1 :).

Now how control flow work that we will see by below example :

```
int main(){
    int num = 8;
    switch(num){
        case 7 :
            printf("value is 7");
            break;
        case 8 :
            printf("value is 8");
            break;
        case 9 :
            printf("value is 9");
            break;
        default :
            printf("out of range");
            break;
    }
    return 0;
}
```

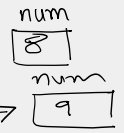
If some statement is given here in this place then it will not get executed.
Switch find check cases if it find match then it execute statement inside case & if it doesn't find case then it will directly jump to default.
If this break is not present then case 9 also gets executed this is called fall-through

NOTE :

1) **In fall-through it will not check it num is matches with case 9 it just executes whatever is there in case 9.**

Let's see another example,

```
int main(){
    int num = 8;
    switch(num){
      default : num++;
      case 2 : printf("Humans are animals");
      break;
    }
    printf("%d", num);
    return 0;
}
```



after default (i.e. num++) num = 9 then it does not check case for 9 it just execute case 2 according with FALL-THROUGH.

In short, we can say,

Switch (. . .) Search for the matching case, if nothing matches then go to default and fall through till break.

Switch(*(14+ "I love quantum" "City")) will result in switch('C').

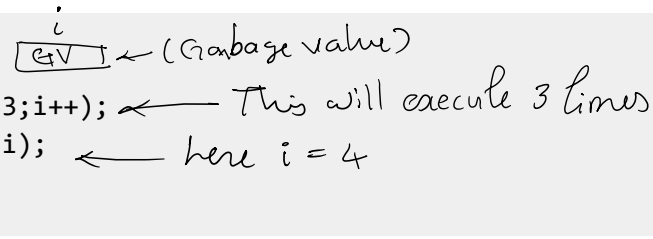
1.3.3) Loops in C-programming :

One thing to make note that as there are three parts of for loops (i.e. initialization, test, condition) and *all of these three parts are optional*.

```
for (initializationStatement; testExpression; updateStatement)
{
    // statements inside the body of loop
}
```

Let's take one tricky example,

```
int main(){
    int i;
    for(i = 0;i<=3;i++);
    printf("%d", i);
    return 0;
}
```



In loops we may encounter break and continue statements. And one thing note that break statement only breaks one loop (while, for, do while). If while loop is nested and inner loop contains break statement then only inner loop gets break. And you very well known to continue statement, continue skips statement after it and it does not end the loop as it name suggests it continues the loop.

1.4) Operator in c programming :

There are many types of operators in c programming some of them are arithmetic operator, relational operator, bitwise operator, etc.

- **Arithmetic operator** :

*, /, %	Multiplication, division and modulo	Left to right
+, -	Addition and subtraction	Left to right

What will be the value of a after executing following operation ? int a = 9 – 12/3 + 3*2 – 1 ⟹ 9 – 4 + 6 – 1 = 5 + 6 – 1 = 11 – 1 = 10

- **Relational operator** :

< <= > >=	Less than / less than or equal to Greater than / greater than or equal to	Left to right
== !=	Equal to and not equal to	Left to right

What will be the output of following program ? –

```
int main(){
    int a = 10, b = 20, c = 30;
    if(c > b > a) printf("True");
    else printf("False");
    printf("%d", 1 == 3 != 5);        //Here The operators == and != have the same precedence. Because their associativity is left to right, the == operation was performed first, if associativity is right to left then 3 != 5 will be implemented first.
    return 0;
}
```

30 > 20 > 10 — True means 1 — 1 > 10 — False

From above example can see that **associativity is not restricted to same operation**.

- **Logical operators** :

!, &&, ||

!	Not operator	Right to left
	. . .	
&&	Logical AND	Left to right
||	Logical OR	Left to right

- **Increment and decrement operators** :

++m; or m++; ⟹ pre-increment and post-increment

--m; or m--; ⟹ Pre-decrement and Post-decrement

Illegal operators in increment and decrement :

1) ++2 – You cannot increment a constant
2) ++(a+b*2) – You cannot increment an expression

- **Bitwise operator** :

Bitwise And : &

```
int main(){
    int a = 13, b = 25;
    printf("%d", a&&b);
    return 0;
}
```

a = +VE
b = +VE
a&&b = 1
This is logical and operator

```
int main(){
    int a = 13, b = 25;
    printf("%d", a&b);
    return 0;
}
```

a = 0000 0000 0000 1101
b = 0000 0000 0001 1001
a&b = 0000 0000 0000 1001 = 9

Bitwise OR : | (same as bitwise *and* but do or operation on each bit)

Bitwise Exclusive OR : ^ (It will give 1bit to each bit combination when exactly one bit is present)

Left shift and Right shift : left shift : op << n ; Right shift : op >> n

Example, x = 0000 0000 0000 0011 if you do x<<2 then x = 0000 0000 0000 1100 basically you are multiplying number by 4 which is 2^2.

Talking about right shift so, suppose x is 0100 1001 1100 1011 and you do x >> 3 then x will be 0000 1001 0011 1001

But there is rule in filling those first 3 bits,

If number is Unsigned – fill zeros

If number is signed – depends on system (either zeros or sign bit)

One's complement operator : ~ (different from logical not operator)

X = 0000 0000 0000 1000 (8)

~X = 1111 1111 1111 0111 (-9)

If you printf("%d", -X-~X) since -X = ~X + 1, it will print 1.

- **Assignment Operators** :

X = 3; and X += 3 (same as X = X + 3)

=	Assignment operator	Right to left
+= -=	Addition/subtraction assignment	
*= /=	Multiplication/ division assignment	
%= &=	Modulus and bitwise assignment	
^= |=	Bitwise exclusive/ inclusive OR assignment	
<<= >>=		

Int a = b = 10; then first b will get value 10 then a = b so a will also get value 10 from b.

In general, v op= exp is same as v = v op (exp) here expression can be number or for example,

x += y+1 is same as x = x + (y+1);

- **Conditional operator or ternary operator** :

exp1 ? exp2 : exp3 — Solve this by this always → if (exp1) exp2; else exp3;

if(--j ? ++j ? i : i++ : i--)

same as → if(--j){ if(++j) i; else i++; }else i--;

- **Comma operator** :

Evaluated left to right and the value of right-most expression is the value of the combined expression.

Value = (x = 10, y = 5, x+y); → x: 10, y: 5, value: 15

t = x, x = y, y = t; ← all three are independent statement
this is different from t = (x, x = y, y = t);

(PoL-PreR-AL-BL-RL-BL-LL-AR)

Operator	Description	Associativity	Precedence
() [] . -> x++ x--	Parentheses or function call Brackets or array subscript Dot or member selection operator Arrow operator Postfix increment/ decrement	Left to Right	1
++x --x + - ! ~ (type) * & Sizeof	Prefix increment/ decrement Unary plus and minus Not operator and bitwise complement Type cast Indirection or deference operator Address of operator Determine size of bytes	Right to Left	2
* / %	Multiplication, division and modulus } arithmetic	Left to Right	3
+ -	Addition and subtraction	Left to Right	4
<< >>	Bitwise left shift and right shift } bitwise	Left to Right	5
< <= > >=	Relational less than/less than equal to rational greater than/ greater than or equal to } Relational	Left to Right	6
== !=	Relational equal to or not equal to	Left to Right	7
&	Bitwise AND	Left to Right	8
^	Bitwise exclusive OR } bitwise	Left to Right	9
|	Bitwise inclusive OR	Left to Right	10
&&	Logical AND } logical	Left to Right	11
||	Logical OR	Left to Right	12
? :	Ternary operator	Right to Left	13
= += -= *= /= %= &= ^= |= <<= >>=	Assignment operator Addition/ subtraction assignment Multiplication/ division assignment Modulus and bitwise assignment Bitwise exclusive/ inclusive OR assignment	Right to Left	14
,	Comma operator	Left to Right	15

NOTE :

1) **C, like most languages, does not specify the order in which the operands of an operator are evaluated. (The exceptions are &&, ||, ? :, and ',') For example, in a statement like x = f() + g(); f may be evaluated before g or vice versa; thus, if either f or g alters a variable on which the other depends, x can depend on the order of evaluation.**
2) **Here one operation has higher precedence over another means that you put brackets in that operation first that's it. It does not represent order of operation. Just brackets that's it.**

1.4.1) Sequence points :

Consider a = 5; x = a++; pf(x) will print 5 and pf(a) will print 6. So, after the semicolon "a" get incremented **or** we can say after the *sequence point* "a" get incremented.

Semicolon is one of the types of sequence point.

Consider, another sentence x = a++ + a++; here a = 5 and after executing a from left hand side we don't know whether second a will have 6 or still 5. It is completely dependent upon compiler. In more formal term, we are trying to modify the value of "a" multiple times before one sequence point (like there are no sequence point after first a++).

So, statements like x = a++ + a++; is useless or you can say answer is undefined.

a++ + a++;
a++ - a++;
a++ * --a;
} Undefined (compiler dependent)

Which means sequence points is nothing but after that point the changes will definitely takes place.

There are other sequence points for example :

Semicolon = ;

Conditional statements = If(), for(), while(), switch()

Ternary operator = ? :

Logical operators = ||, &&

```
i = 5;          6
if(i++) { a = i;
a = i++ ? i : i--
a = i++ || i--
```
} Valid

If these are given after increment or decrement then changes will definitely take place.

1.4.2) Short circuiting, order of evaluation and associativity for logical and/or :

```
int a = 1, b = 1, c = 1;
    if(a==b || c++) printf("%d", c);
```
Output : 1

After executing a == b which gives value 1 (true), true || anything is true. So c++ will not get executed. This is called **short circuiting**.

Question :

1) Predict the output

```
int j = 1, var = -(-j);
printf("var=%d", var);
```

both are unary op

this combined will give 1

this will give -1

So var = 1

```
int var = -(-3);
printf("%d", var);
```

space is ignored

var = -(-3) = 3

var = 3 printed.

1.5) Functions in C programming :

If you want to do some task again and again and writing same code will consume time and space. For example, printf(" ") is a function which prints the values, which consists of large code (not visible to

user) but it is so often that we create something called function (easier to manage in smaller pieces, like functions). Same as functions in mathematics y = f(x). Syntax of function is given below,

```
function_type function_name(parameter_type parameter_name){
    local variable declaration;
    executable statement1;
    executable statement2;
    .....
    return statement;
}
```

Return type; function name; function body; This is definition not declaration; curly brackets to group statements into blocks.

Equally acceptable forms of **declaration**

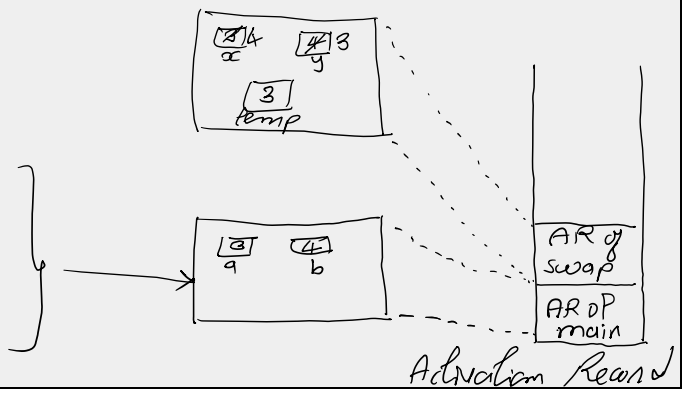
```
int mul (int a, int b);
int mul (int, int);
    mul (int a, int b);
    mul (int, int);
```

By default return type are integers

In older version of C (example C88, C89, C90, C95) if function returns integers then declaration was optional. But in Latest version of C we have to declare the function before calling it. If we do not declare before calling in latest version it will throw error as "*error: 'function_name' was not declared in this scope*".

Let's see the flow of program (which include function) and how it's getting executed internally

```
void swap(int x, int y){
    int temp = x;
    x = y;
    y = temp;
}
int main()
{
    int a = 3, b = 4;
    swap(a, b);
    printf("a = %d, b = %d", a, b);
    return 0;
}
```



After calling swap(a, b); in main AR of swap gets pop from Activation record stack and as you can see it is not changing a and b. so at output, we get a = 3, b = 4.

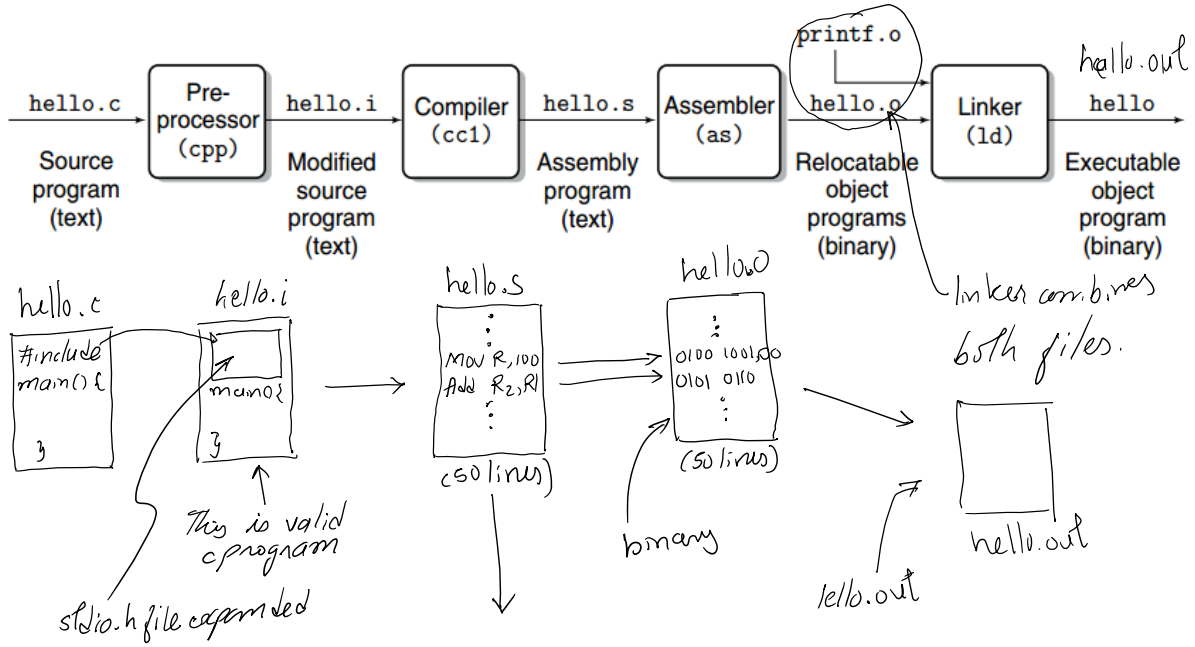
```
int easy(unsigned int x, unsigned int y){
    if(x-y>0)return 1;
    else return 0;
}
int easy(unsigned int x, unsigned int y){
    if(int(x-y)>0)return 1;
    else return 0;
}
```

Resultant output is considered unsigned

here first it is unsigned & then we convert it into signed

In above one thing to note that x-y is same as x + 2's complement of y. So do not calculate think it in terms of computer or compiler.

A journey of the program : from writing to running



We basically write code and we want hello.out file at output so consider one scenario that we have complied file but we want to run that file after 5 years. And Consider we have mov R, 100 instruction, which tells that go to 100 location and copy that data to Register. So, after 5 years how does compiler know which 100 location because it may happen that those data can be overwritten by some other program in memory. So, because of this we use concept of logical memory here. Compiler don't even care about physical memory. It assumes all the (logical) memory is available to just one process and generates logical address (relocatable).

After COA watch c programming – module 2 – 5d

1.6) Storage classes in c program :

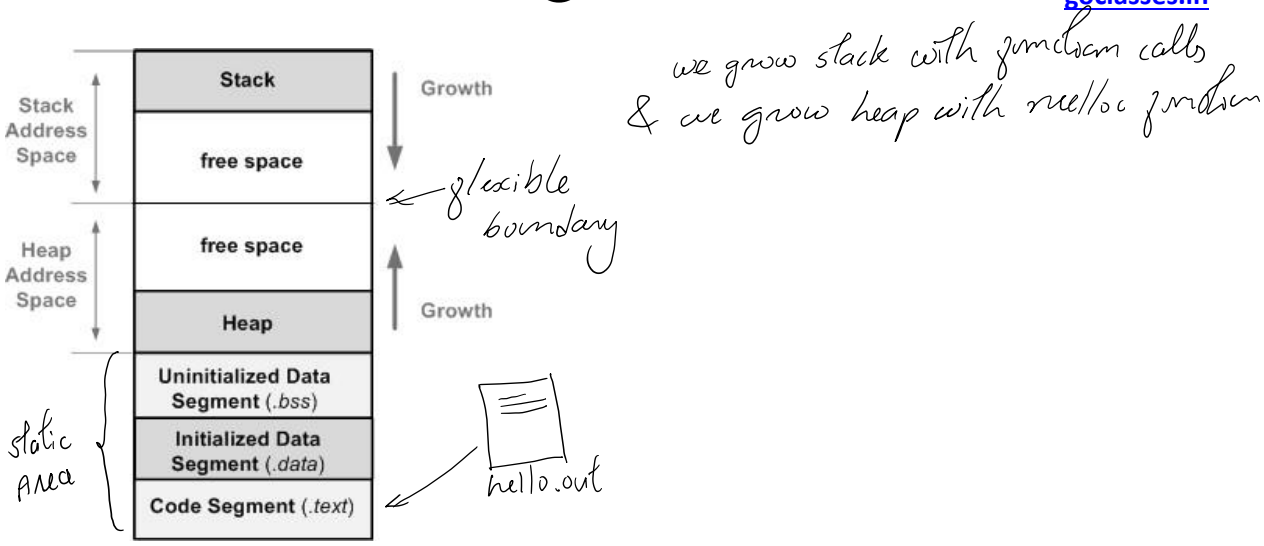
There are four basic storage classes in c program : auto, register, static, extern. (local variables; for local & global)

1.6.1) Memory layout of a program :

We know that when we write code after compilation (here compilation means from hello.c to hello.out) we have to load hello.out in memory. By default, compiler will store it in virtual memory. Memory layout of a program is given below.

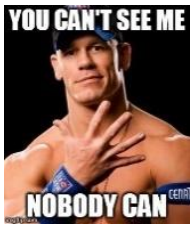
```
void moo(char* to_print)
{
    char* cheerful = "printf is awesome\n";
    printf("%s", to_print);
}
```

At compile time, the compiler checks the code to ensure that it is syntactically correct and generates machine code based on the source code. During this phase, the number of arguments that moo() passes to printf() is known.



1.6.2) auto - storage classes :

Before we go ahead, we need to understand few terms like scope and lifetime.



Scope : can you see me ?

```
{ int a = 5;
}
printf(a)
```
this is one block
compile time error

Lifetime : are you alive (in memory) ?

```
{ int a = 5;
}
printf(a)
```
This a is not in memory

Can you see me ?	Are you alive (memory)?	Possibility	Example
✅	✅	Possible	{ int a = 5; } ← at this point
❌	❌	Possible	{ int a = 5; } ← at this point
✅	❌	Impossible	Not even in memory so how it is visible?
❌	✅	Possible	{ static int a ; } ← can not access

Now, let's take back our attention to auto storage class, Basic info about auto is given below :

- Storage area – Stack at runtime
- Initial value – Garbage
- Scope – within block
- Lifetime – end of block
- Only used for local variables

The auto storage class is the default storage class for all local variables. Now, linker links more than one files and we also know that auto variables are not accessible outside the block so, how can it be possible to access it outside the file ? – Not possible. That is why,

Objects with the auto class are NOT available to the linker.

```
auto int a;                    /*illegal - auto must be within a block*/
main (){
```

```
    auto int b;                 /*valid auto declaration*/
    for(b = 0; b<10; b++){
        auto int a = b + a;     /*valid inner block declaration*/
    }
}
```

1.6.3) Register storage class :

This is same as auto but variable may store in cpu register.

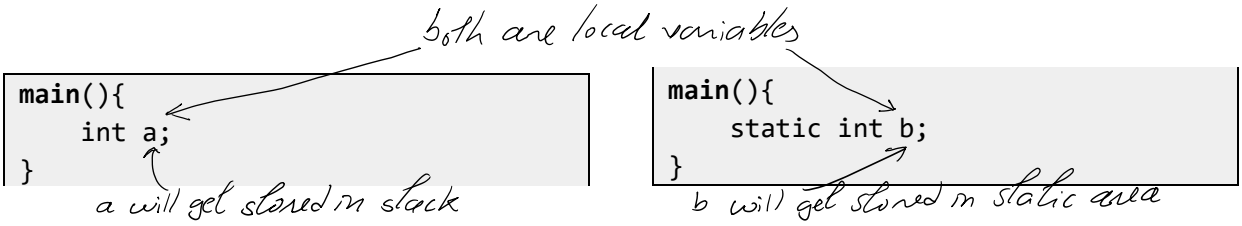
- Storage – CPU register
- Initial value – Garbage
- Scope – Within block
- Lifetime – End of block

We know that registers are superfast because they are easily accessible by CPU. So, when we write register in front of any variable it's a hint to the compiler that the variable will be heavily used and that you recommend it be kept in a processor register if possible. Most modern compiler do that automatically, and are better at picking them than us humans. That is why no one use register keyword. And no one use auto because by default every variable within block is auto.

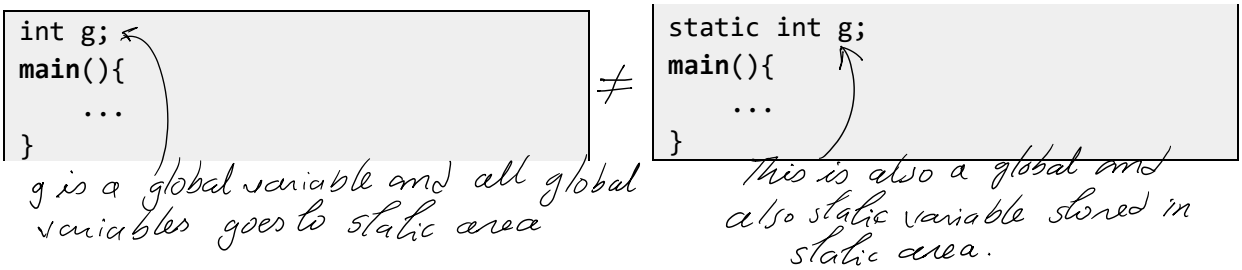
You can't use auto or register for global variables

1.6.4) Static Storage class :

- Storage – Static memory at compile time
- Initial value – Zero
- Scope – Within block
- Lifetime – Till end of main program (It get destroyed only if program terminates)
- Objects with the static class are not available to the linker.



Auto is default keyword for local variables.



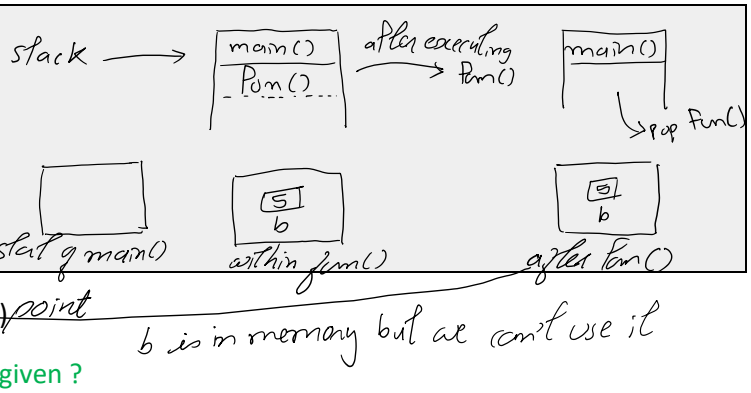
Static is not a default keyword for global variables.

Although both are stored in static area but both are different.

The difference is that you can see global “int g” outside the file (means you can use that g to another file) but you cannot use “static int” global variable g outside the file. And that is why static class are not available to the linker.

So, we can specifically say that scope of static = within block (if local than within block and if global then within same file meaning you cannot access that variable in another file)

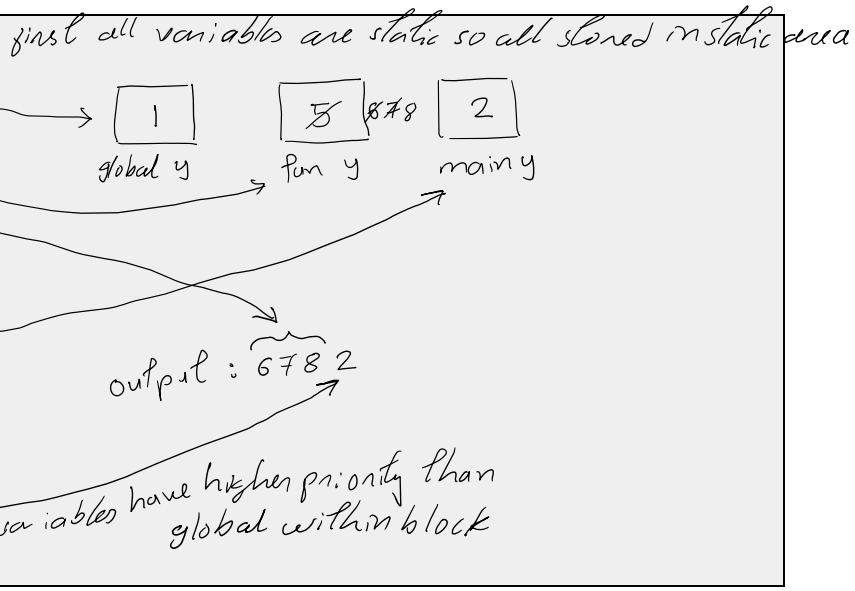
```
fun{
    static int b = 5;
}
main(){
    fun();
    .
}
```



This is “can you see me” (forth example)

Q : What will be the output of program given ?

```
#include<stdio.h>
static int y = 1;
void fun(){
    static int y = 5;
    y++;
    printf("%d", y);
}
int main(){
    static int y = 2;
    fun();
    fun();
    fun();
    printf("%d", y);
    return 0;
}
```



The initialization of static variables can only happen at compile time.

```
int main(){
    int i = 10;
    static int x = i;
    printf("%d", x);
    return 0;
}
```

This will lead to compile time error as i is auto variable whose value is determined at runtime but static variable can only be initialized in compile time.

Why such weird rule ? – This is simple because int i = 10; this value 10 will be first load into main memory which can only happen at run time but for static variable we need not access main memory we simply write it in static storage.

1.6.5) Extern storage class :

- Storage – static memory
- Initial value – Zero

- Scope – Global
- Lifetime – till end of program
- Objects with the extern class are available to the linker.

If we write extern int max; It is a way to tell compiler that there is a **global** variable max. compiler does not create variable max on executing extern keyword. More specifically take an example :

```
#include<stdio.h>
int main(){
    int len;
    extern int max;
    printf("%d", max);
    max = 5;
}
int max;
```

This line is telling to the compiler that - Please don't worry, There is a global variable max somewhere in this file or in another file.

output : 0

But in previous example as you can see, if we write int max after #include line then there is no point of writing extern int max because compiler will obviously know that some max variable is there (global or local) and it'll act accordingly.

So, common practice is to place definitions of all external variables at the beginning of source rule and them omit all the extern declarations.

Q : Is this program gets any compilation error or runtime error (only this file is present) ?

```
#include<stdio.h>
int main(){
    extern int max;
    max = 5;
    printf("%d", max);
}
```

there is global variable max in this file or in another file (Don't worry)
so no, compilation error but at the time of linking linker will not able to find max so at runtime we get linker error.

So, when linker error will not occur ? – when you have specified extern keyword and that variable is used somewhere and it is present in some other file or in present file then no linker error. And one more case is possible which is when you have specified extern keyword for some variable but that variable is not used anywhere then linker will ignore such line.

If some function is declared but not defined then that program will successfully compile but during the linking phase, the compiler tries to resolve function calls to their actual definitions. If it cannot find definition then it will produce error as function is undefined.

Compiler checks declaration and Linker checks definition

```
#include<stdio.h>
int main(){
    extern int i = 10;
    printf("%d", i);
    return 0;
}
```

This will throw compilation error
No meaning, extern used to specify compiler that such global variable exists

```
#include<stdio.h>
extern int i = 10;
int main(){
```

if you are using it outside the function
that means extern int i = 10; ≈ extern int i;
i = 10;

```
    extern int i;
    printf("%d", i);
    return 0;
}
```

In this there is no error
output : 10

NOTE :

1) **You cannot write anything other than declaration or definition outside functions. For example, writing i = 5 outside the main (globally) will give compile time error.**
2) **By default, functions are extern.**

How to link more than one c\cpp files ? – first create three files for example, a.cpp b.cpp and c.cpp then go to that file directory in command prompt then type "*gcc -o main .\a.cpp .\b.cpp .\c.cpp*", hit enter and execute it in main files by command ".\main.exe" hit enter again. Second method is to use #include "file_name" this command we have to write in main file and then simply run the program.

2. RECURSION, POINTER and ARRAY

Loopless loop – Functions defined in terms of itself are called *recursive functions*.

2.1) Introduction to recursion :

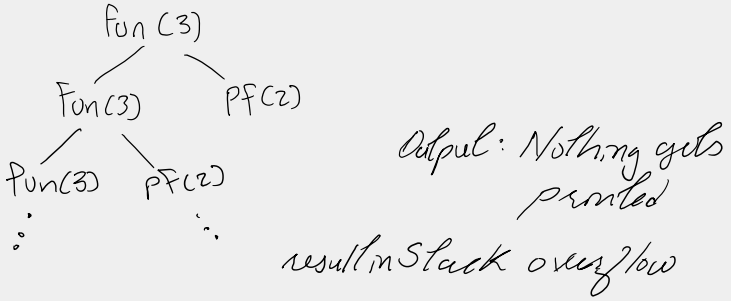
You know this concept very well just one example is enough which is factorial function.

Q : How does it work internally ? – First in stack main() function will get pushed then all the recursive call are pushed. Now, what do you mean by recursive calls are pushed is it code that is pushed or something else ? – Activation call is being pushed on to the stack not code because code have its own area in static. For our simplicity we use tree diagram to solve problem.

```
int recsum(x){
    if(x==1) return x;
    else return x + recsum(x-1);
}
```

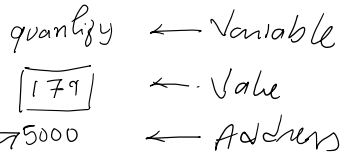
Take another example,

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



//Module 3 - Lecture 8a

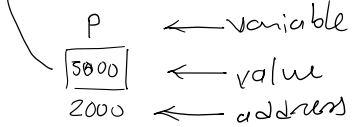
2.2) Pointers and Array :



Pointers allows us to indirectly access variables.

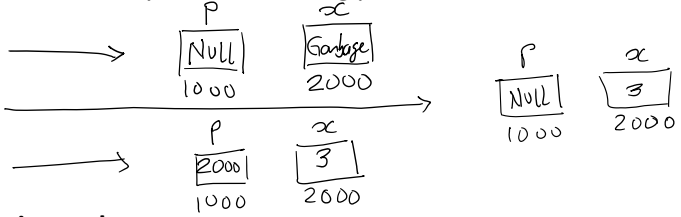
Int quantify = 179;

This is something similar to having a house number as well as house name. Pointer is nothing but a variable contains an address.



int *p = &quantify;

Let's have a look at memory while executing pointers,



Pointer declaration style :

Int* p;

Int *p; ← perfect

Int * p;

Some Illegal uses of pointers :

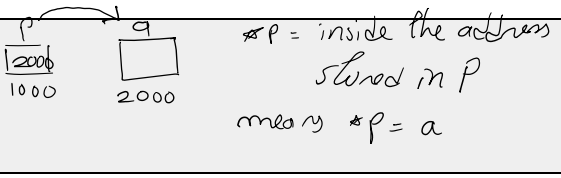
- &125 (pointer at constants).
- &(x+y) (pointing at expressions)
- Register int x; int *p = &x; (You cannot take an address of register variables, even if compiler decides to keep variable in memory rather than in register).

//Module 3 – Lecture 8b

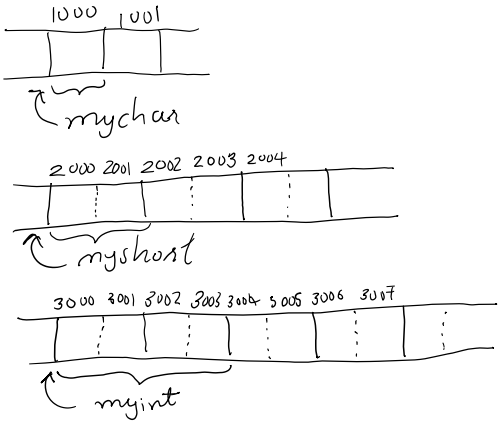
2.2.1) Accessing the variable through the pointer :

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

equivalent to b = a
at this point *p = a



which means *(&p) = p and &(*p) = p.

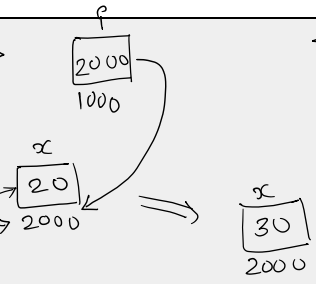


char *mychar;
short *myshort;
int *myint;

//Module 3 – lecture 8c

2.2.2) Passing pointer to function : pointers are useful when we want to change local variable value in some another function. Example,

```
change(int *p){
    *p = *p + 10;
}
main(){
    int x;
    x = 20;
    change(&x);
    printf("%d", x);
}
```



*p = *p + 10
x = x + 10
x = 30

2.2.3) Introduction to array :

Array is a data structure which can represent a collection of data items which have the same data type (float/int/char/…)

All the data items constituting the group share the same name

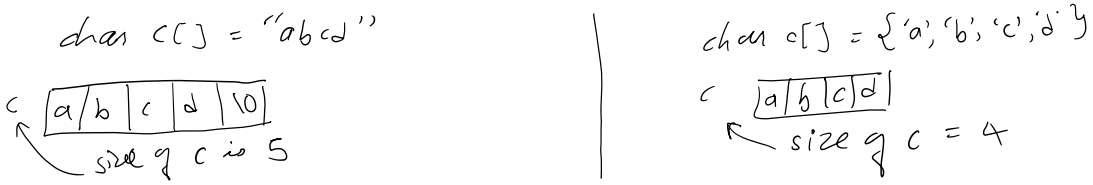
Int x[10]; ← group of integer

Array initialization :

```
int myArray[10]={5,5,5,5,5,5,5,5,5,5};
int myArray[10]={1,2};      //initialize to 1, 2, 0, 0,...
int myArray[10]={1};        //first element is 1 and all other element 0, 0,...
int myArray[10]={0};        //first element is 0 and all other element 0, 0,...
static int myArray[10];     //all element 0
```

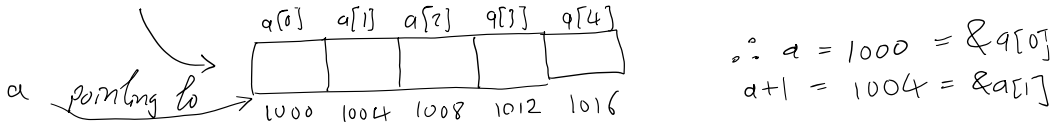
In case of character array default value is null character '\0' (ASCII value zero) like in array we have only specified {1, 2} and compiler has automatically considered 0 value for rest like that in character default value is '\0'.

If dimension is not specified the compiler will deduce the dimension from the initializer list. For example, int arry[]={1, 2, 3, 4, 5} here size of array is 5.



2.3) Array and pointer together :

If we have int a[5]; then we know it is group of 5 integer but what is a ?

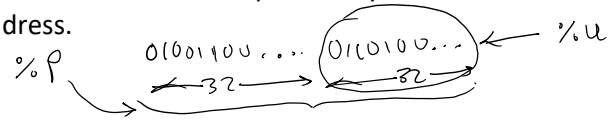


a[i] –

- a represents address of first element in array
- a+0 represents address of first element in array
- a+1 represents address of second element in array
- a[i] is same as *(a+i) which means if we write i[a] then also it is correct representation of *(i+a).

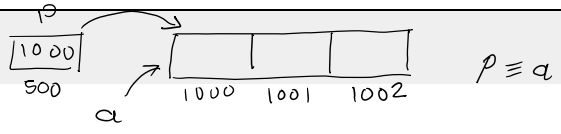
Q : Format specifier for address is %p and sometimes we use %u but what is the difference ? –

%p = designated format specifier and u% is for unsigned integer. We can use %u as address is always unsigned and it is integer but suppose address is 64 bits and integer are of 32 bits. Then when we use %u it will ignore first 32 bits of address and prints only last 32 bits of address. But if you use %p instead we will get correct address.



Now, consider following code :

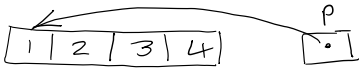
```
int *p;
int a[3];
```



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

Q : What is the difference between p and a? –

- 'a' cannot be used on left hand side i.e. a = a+n (illegal), but we can use for 'p'
- P has its own box or storage, a doesn't has its own storage.
- Sizeof p is size of address (let's say 8bytes) but sizeof (a) = 3*sizeof(int) = 3*4bytes = 12 bytes. Because p is representing address whereas a is representing whole array.
- Whatever type of pointer can be p (like char *p, long int *p,…) as it represents address, sizeof(p) always gives same result. While char a[10] and sizeof(a) then it will give 10 (bytes).

Now, consider an array – a : 

Different ways to access particular element of array :

- a[3]
- 3[a]
- *(a+3)
- P[3]
- 3[p]
- *(p+3)

2.3.1) Size of operator :

The size of operator is the only one in c, which is evaluated at compile time. Whereas +, -, or any other operator are evaluated at runtime because they need CPU for calculation. Let's understand this by an example,

Int I=1; printf("%d", sizeof(i++)); printf("%d", i); output is 41 why 1 because at the execution of sizeof(i++) as we know that sizeof operator is evaluated at compile time and i++ is evaluated at runtime but it got no chance and compiler just evaluated sizeof(int) as I is integer. So output is 4 and 1. Thus,

Compiler doesn't evaluate any expression inside *sizeof* operator

Some interesting things happen when we use addition inside sizeof operator. Consider lines of codes below :

```
int a;
long int b;
printf("%d", sizeof(a+b)); //output : 8 (long int promotion)

int a;
char b;
printf("%d", sizeof(a+b)); //output : 4 (char to int promotion)

char c = "d";
char d = "e";
printf("%d", sizeof(c+d)); //output : 4 (char to int promotion)
```

This is exception

2.3.2) Pointer arithmetic :

If we have p as pointer then (p+1) is implemented by assembler as p + sizeof(one element) this sizeof(element) depends upon data type of variable stored at location at which p is pointing to. Take an example,

```
int main(){
```

```
    char s[]="abcdefgh";
    char *t = s;
    int *p = (int *)s;

    printf("t = %u\n", t);
    printf("t+1 = %u\n", t+1);

    printf("p = %u\n", p);
    printf("p+1 = %u\n", p+1);
    return 0;
}
```

type casting unless it will show error

1 byte increment

4 byte increment

Output : t = 6422287

t+1 = 6422288

p = 6422287

p+1 = 6422291

both points to same address but increment happens based on data type of pointer to which they are pointing.

so, int *p p+1 = p + sizeof(int)

char *t t+1 = t + sizeof(char)

long int *z z+1 = z + sizeof(long int) … In short, we can say **q + 1 = q + sizeof(*q)**

Any pointer can point anywhere

Pointer can point to outside of the array also and it is completely valid but it will give garbage value. If a[3] = {1, 2, 3} and now you are accessing a[9] then it is completely valid. But results in garbage value.

2.3.3) Valid arithmetic pointer operations :

Generally, we do pointer arithmetic only on elements of arrays

- **Adding or subtracting an integer to the pointer**
- **Subtracting two pointers from each other** (tells the distance in terms of number of elements)

When two pointers are subtracted, both shall point to elements of the same array. *If not then result can be any garbage value or error.*

Not restricted to pointers, address also application

While subtracting two pointers, internally it is executed as $p_2 - p_1 = \frac{p_2 - p_1}{sizeof(*p_2)} = \frac{p_2 - p_1}{sizeof(*p_1)}$ that is why we say it is equal to elements of the same array objects.

We can have pointers to whole array in that case p2 – p1 gives total number of arrays between p2 and p1.

- **Comparing pointers** (p>q, p!=q, p<q+4, …)

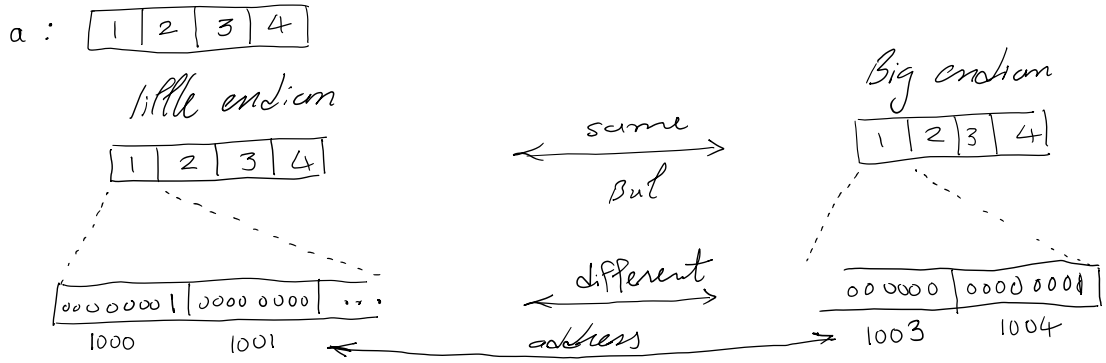
Invalid arithmetic on pointers :

P1 + p2, p1 * p2, p1 % p2, p1 / p2,…

// Lecture 10a

2.4) Char pointer, Double pointers and 2D array :

Int a [] = {1, 2, 3, 4} endianness does not alter data of array. But it alters binary representation of integer.

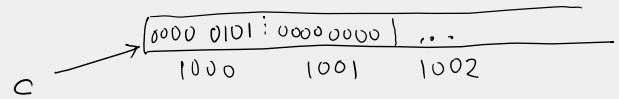


Q : How to know which system our compiler is using ? – Consider following example,

```
}
```

So, while converting integer to binary we follow big endian and then while calculation in code we use little endian.

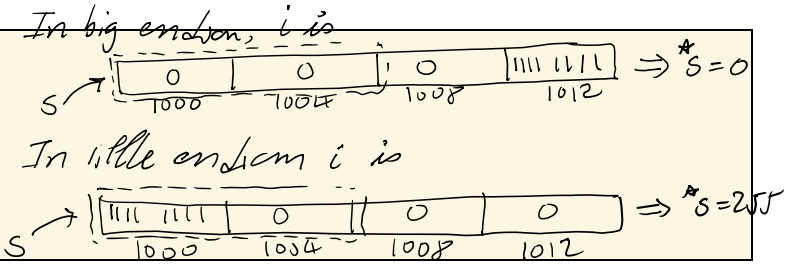
```
#include<stdio.h>
int main(){
    int i = 5;
    char *c = (char *)&i;
    printf("The first byte = %d\n", *c);
    printf("The second byte = %d\n", *(c+1));
    printf("The third byte = %d\n", *(c+2));
    printf("The fourth byte = %d", *(c+3));
    return 0;
```



little endian

output : 5
0
0
0

```
int main(){
    int i = 255;
    short *s = (short *)&i;
    printf("%d\n", *s);
    return 0;
}
```



But in little endian *s will be again gets reverse due to little endian

So, *s = 0000 0000 1111 1111 = 255 (in decimal)

In C language only void and char pointers are allowed to point to other types even though all pointers have the same size.

// Lecture 10b

2.4.1) Passing 1D array to the function :

Logic is very simple, if you are passing some address then there should be some pointer or same structure to hold that address.

compiler will convert it into int *p (it will ignore k)

```
void fun(int p[k]){
    p++;
    printf("%d", *p);
```

k is any non-negative integer

```
}
main(){
    int a[3]={1, 2, 3};
```

```
    fun(a);
}
```

Output : 2

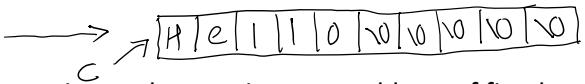
If k is negative then compile time error

//Lecture 10c

2.4.2) Character array and strings :

First of all, there is no string datatype in C programming, we store it in character array.

char c[10] = "Hello"



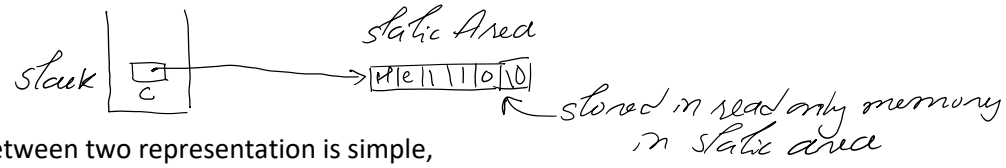
To print while whole array as string we have to just pass address of first byte.

printf("%s", c+1); --- first it will go to c+1 i.e. e and printf character till "\0" special character.

String literals in C programming :

char *c = "Hello";

In this representation, we have pointer c in stack which points to character stored in static area.



The difference between two representation is simple,

char c[] = "hello"; --- c[0] = 'g'; meaning you can do modification in this array but

char *c = "Hello"; --- this is treated as constant and we cannot modify this (ROM in static area)

```
char a[] = "abc\0de";
char  *p = a ;
```

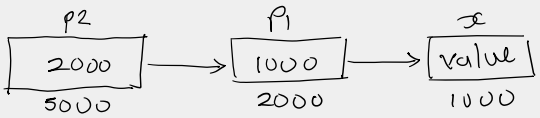
```
sizeof(a) = 7
sizeof(p) = 4
```

```
sizeof(*p) = 1
strlen(p) = 3
```

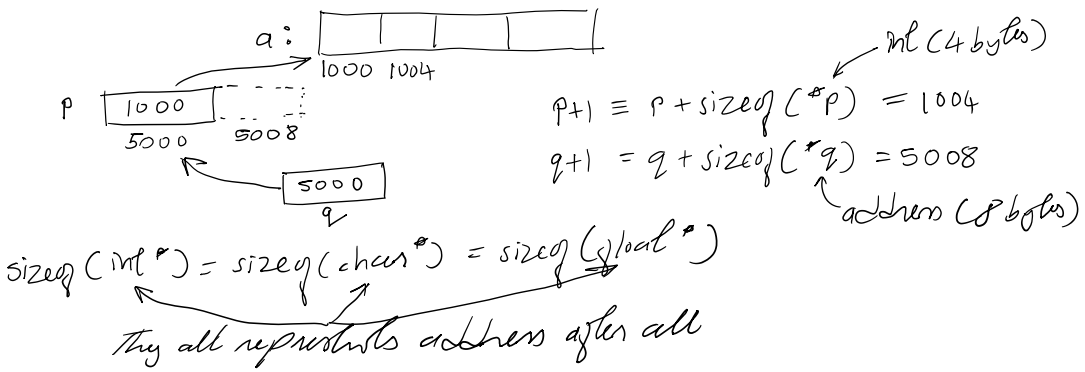
// Lecture 10d

2.4.3) Double pointer : pointers to pointers to…

```
int x, *p1, **p2;
x = 100;
p1 = &x;
p2 = &p1;
printf("%d", **p2);
```

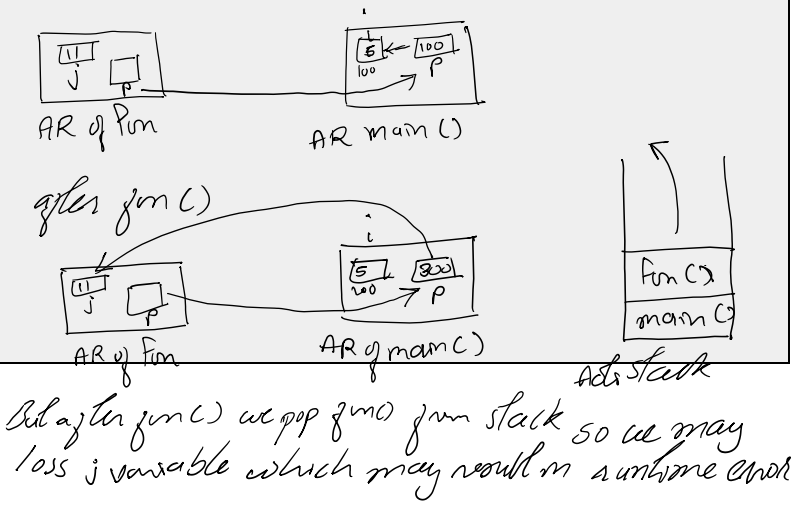


Consider following scenario,



Consider one beautiful example of behavior of pointer :

```
#include<stdio.h>
void fun(int **p){
    int j = 11;
    *p = &j;
    printf("%d", **p);
}
main(){
    int i = 5;
    int *p = &i;
    fun(&p);
    printf("%d", *p);
}
```

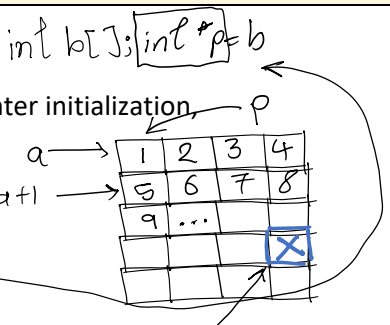


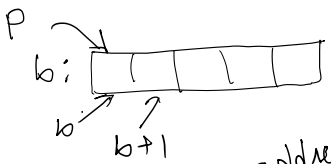
But after fun() we pop fun() from stack so we may loss j variable which may result in unknown error

2.5) 2D array in C programming :

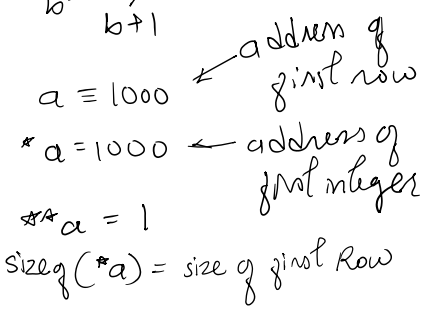
```
#include<stdio.h>
int main(){
    int c = 10;
    int *k = &c;
    int b[] ={10, 11};
    int *q = b;
    int a[5][4]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17,
18, 19, 20};
    int (*p)[4] = a;
    p++;
    printf("%d ", **p);
    printf("%d ", *q);
    printf("%d", *k);
    return 0;
}
```

Till now we have seen that,





Now, consider 2D array and its pointer initialization,



In 2D array, p is pointing to whole array of size 4.

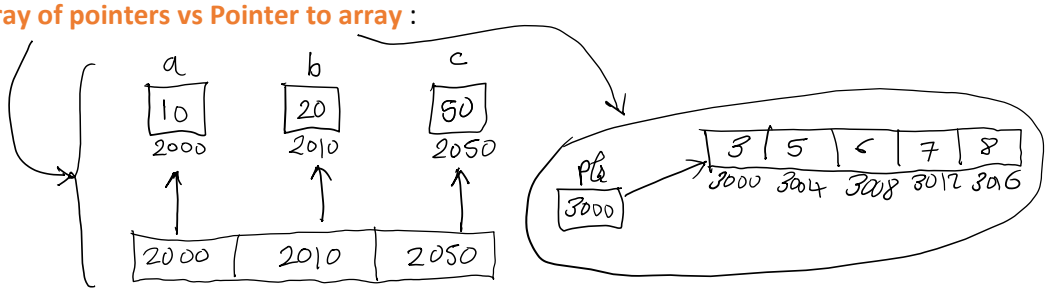
Different ways to access element "x" :

- First, we will go to forth row so, a+3 then we go inside that row to access array. So, *(a+3). Then we will go to that element address by adding +3 thus, *(a+3)+3 then we access that element so, *(*(a+3)+3)).
- Now, we know that *(a+3) is just a[3] so, *(a[3]+3) is second method and we again apply same method to get a[3][3].
- But wait there is another method, we know that memory is continuous and a contains base address and first element is at base address. And we observe that by counting x is at 15th index so, *a+15 will get us to x mark but not inside of x element. So, we do *, *(*a+15). Now, *a = a[0] which means *(a[0]+15) again applying we get a[0][15].

In short, we have discussed five ways to get x namely *(*(a+3)+3), a[3][3], a[0][15] and a[1][11] (how?) and *(*(a+4)-1) (how?)

//Lecture 11b

2.5.1) Array of pointers vs Pointer to array :



1) **Pointer to array** :

Int (*p)[4]; ← p is a pointer to an integer array of size 4. But why parathesis ? What if we remove parathesis ? – then it will become int *p[4] ← first brackets have higher precedence so it will declare array p of size 4 then it will take * which means it is array of pointers.

NOTE :

1) **Don't think & and * as address and inside array or something like that. Don't act fool. Always remember foundation. i.e. & is reference meaning if we have array a, so here a contains address of first element of array but when you apply &a then it is saying come out from address of a meaning, we are now pointing to whole array.**
2) **There is a difference between obtaining address of some memory location and obtaining values of some memory location. First will not give you error but obtaining or accessing values of some unknown memory location will give runtime error.**

```
int a[]={16, 2, 3, 4, 5};
int *i = a+1;          //here i is just a pointer which is storing add. Of 2nd element of array
int (*ptr)[5] = &a;    //here ptr is pointer to array so we don't write just a if we write then it is address of first element but we want to represent whole array so we come out from element to array by applying &.
printf("%d %d", *(i-1), sizeof(*ptr));
```

- Call-by-name works like a macro and substitution happens only during use time. For example, if we pass 2+3 to the below function

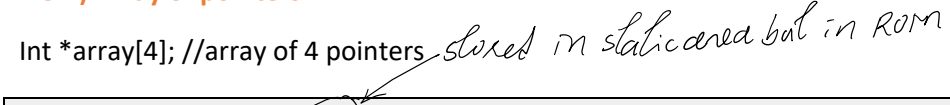
```
int foo(int x){
    return x * x;
}
```

We get 2+3*2+3 which will be 11 due to the higher precedence for *. But, call by reference will return 5*5 = 25 (for call by reference, when an expression is passed, a temporary variable is created and passed to the function).

//Lecture 11c

2.5.2) Array of pointers :

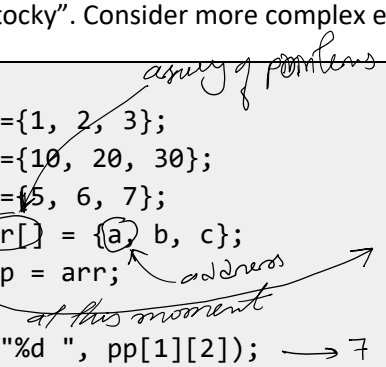
Int *array[4]; //array of 4 pointers

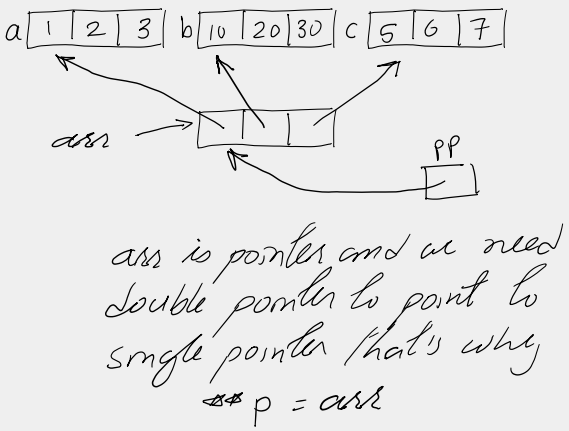



```
char *sports[5]={"golf","hocky"};   //valid declaration of array of pointers
sports[1][0] = 't';                 //Trying to write in ROM (String literals)
printf("%c", sports[1][0]);         //valid access
```

but instead of array of pointers, 2D array would be there then second line will be executed and hocky will become “tocky”. Consider more complex example,

```
int main(){
    int a[]={1, 2, 3};
    int b[]={10, 20, 30};
    int c[]={5, 6, 7};
    int *arr[] = {a, b, c};
    int **pp = arr;
    pp++;
    printf("%d ", pp[1][2]);
    printf("%d ", **(pp++));
    printf("%d ", (**pp)++);
    printf("%d ", **pp);
    return 0;
}
```





So, need to do three things while solving question related to pointers :

1) *Make diagram*
2) *Understand type of pointers (whether it is array of pointers or simple pointers)*
3) *While printing value, solve operation according to the formate specifier.*

//Lecture 11d

2.5.3) Passing 2D array to the function :

```
fun(int p[5][6]){

    ...
}
```

```
int main(){
    int a[5][6];
    fun(a);
}
```

We passed **int p[5][6]** but internally compiler will convert it as **int (*p)[6]** it means we can even modify pointer p like p = p+1;

Other valid and invalid representation :

Fun(int a[][y]) ✓ Fun(int **a) ✗

Fun(int a[x][y]) ✓ Fun(int a[x][]) ✗

//Lecture 13a

2.5.4) Reading complex declarations :

Translation to English :

* as "**pointer to**"

[ ] as "**array of**"

( ) as "**function returning**"

(int) as "**function taking integer as argument/parameter and returning**"

Step 1 : Find the identifier. This is your starting point. Then say to yourself, "*Identifier is*." You've started your declaration.

Step 2 : Look at the symbols on the right of the identifier. If, say, you find a "[ ]" there, you would say, "*identifier is array of*". Continue right untill you run out of symbols **OR** hit a **right** parenthesis ")". In case if you find ")" – See Step 3.

Step 3 : Keep going left (and keep translating to English) until you run out of symbols **OR** hit a **left** parenthesis "(".

Let's take example, int (*p)[10]; - According to step 1st here p is identifier so we say "P is". Then we resolve () brackets or parenthesis so we say "P is pointing to". Then again go to [10] so, "P is pointing to array of 10". And we reached dead end so we go to right and start reading data type. In the end "P is pointing to array of 10 integer".

If () brackets were not given then we will continue reading to the left. Note that we will do not go to right to say P is pointing to.

- int (*pf) () → Pf is pointing to function returning integer.
- Int (*pf) (int) → Pf is pointing to function taking int as parameter and returning integer.
- Int (*f) (int *) → f is pointing to function talking pointing to integer and returning integer.
- Int *(*(*fp1) (int)) [10] ; → fp1 is pointing to function taking int as argument and returning a pointer to an array of 10 integer pointer.
- Int (*apa[5])[10] → apa is array of 5 pointers to array of 10 integer.
- Int*((a())[])() → a is function which returns array of function which returns pointer to int

Pointer arithmetic : Combining * and ++/--

- ***p++ OR *(p++)**; value : *p, inc : p
- **(*p)++**; value : *p, inc : *p
- **++(*p)**; value : (*p)+1, inc : *p
- ***++p**; value : *(p+1), inc : p

//Lecture 13d

2.6) Types of pointer and dynamic allocation :

2.6.1) Void pointer :

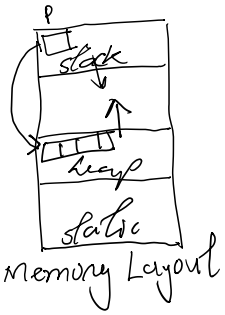
Declaration : void *p = ... ← we can supply any address of data type

But it is special type of pointer. As we don't know to which it is pointing to it may be 2D array or single variable. So, we cannot directly dereference the void pointer, Means we cannot just printf("%d", *p). Before dereference, we need to typecast. For example,

```
int x = 7;
void *p;
p = &x;
printf("%d", *((int *)p));
```

It is very obvious that size of p remains constant i.e. size of pointer as it is pointer after all.

2.6.2) Malloc and free :



Int *p = malloc(4); … Here 4 represents no. of bytes

Above code represents we have asked for 4 bytes from the heap memory.

We can also take variable but n must be defined. Ex. **Int *p = malloc(n);**

int * p = (int *) malloc (1);

*p = 3;

In above code you are first assigning 1 byte and then trying to store number which can take 4 bytes of memory. As 3 is integer and will take 4 bytes of memory but only 1 byte of memory is allocated. In such case either heap will extend the memory or it may not because of possibility of presence of another data. So, this program may crash.

Which means if you allocate any byte of memory randomly by melloc call it doesn't know how you are going to use this memory. And that is why return type of malloc is void*.

- Malloc returns the void type pointer (base address of region), we can use that without typecast.
- On fail, malloc returns NULL

Releasing the used space : Free

free (ptr); It is a way to tell OS that we (programmer) don't need the memory pointed by ptr.

If we try to access *p after free statement, program may crash. But after free, ptr is still pointing to heap address (meaning it still holds address of heap) it is that we **cannot access it**.

//Lecture 13e

2.6.3) Memory leak and dangling pointer :

1) Memory leak :

A memory leak is memory which hasn't been freed, there is no way to access (or free it) now.

```
void fun(){
    char *ch = malloc(10);
    //free(ch);
}
main(){
    fun();
    //ch not valid outside, no way to access malloc-ed memory
}
```

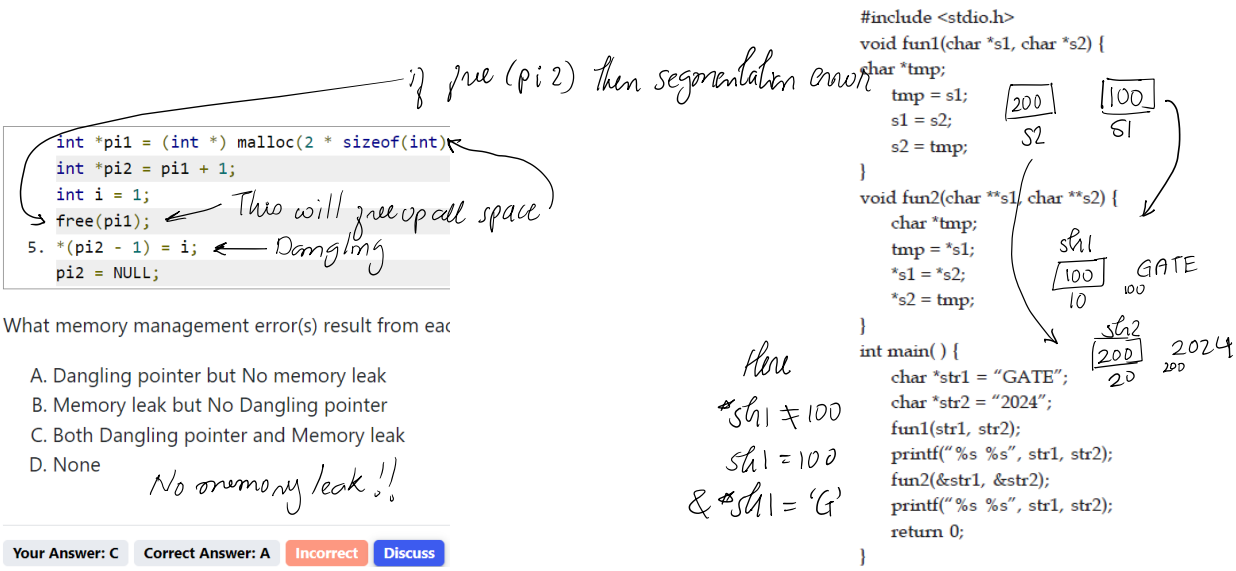
But if we would have written free(ch) then we did not have memory leak problem.

2) Dangling pointer :

A *dangling pointer* **points to** or accessing memory that has already been freed or no longer valid.

```
char *fun(){
    char str[10] = {'H', 'e', 'l', 'l'};
    return str;
}
main(){
    char *c;
    c = fun();
    *c = 5; // Dangling pointer }
```

Silly mistakes :



3. STRUCTURE IN C

3.1) Introduction :

We know that array can store only one type of data. But what if we have to create data structure which stores multiple data type. This is when we use structure.

Structure is used for handling a group of logically related data items. Example : student name, roll number, and marks. Real part and complex part of a complex number.

Helps in organizing complex data in a more meaningful way. The individual structure elements are called *members*.

Syntax :

```
#include<stdio.h>
struct Box{
    int width;
    int length;
    int height;
};
struct Circle{
    double radius;
};
int main(){
    struct Box b;
    struct Circle c;
}
```

Declaring struct variables :

Method 1

```
struct employee{
    int id;
    char name[50];
    float salary;
};
struct employee e1, e2;
```

Method 2

```
struct employee
{
    int id;
    char name[50];
    float salary;
}e1, e2;
```

Method 3

```
struct employee{
    int id;
    char name[50];
    float salary;
}t;
main(){t s = {1, "c", 2.3}
```

Typedef : typedef is a way in C to give a name to a custom type.

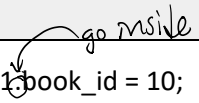
```
typedef type newname;
```

```
typedef int dollars;
typedef unsigned char bytes;
//we can declare variables like :
dollars d;
byte b, c;
```

we can also apply these typedef in structure if you see In method 1 we have used struct employee e1, e2; so, if we want to make one struct employee we have to write these whole sentence we can reduce it using typedef. For example.

```
struct bk{
    char title[50];
    char author[50];
    char subject[100];
    int book_id;
};
typedef struct bk Books;
Books book1, book2;
```

```
typedef struct {
    char title[50];
    char author[50];
    char subject[100];
    int book_id;
}Books;
Books book1, book2;
```

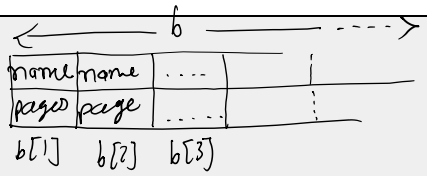


To access any variable, we write book1.book_id = 10;

3.2) Pointer to struct :

Array of structure :

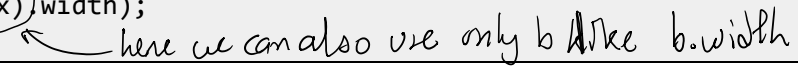
```
struct book{
    char name;
    int pages;
};
struct book b[10];
```



Note that here b is not pointer or something like we saw in previous chapter. Here it is only name.

Pointers :

```
struct Box{
    int width;
    int legth;
    int height;
};
int main(){
    struct Box b;
    struct Box *pBox;
    pBox = &b;
    b.width = 3;
    printf("%d", (*pBox).width);
}
```

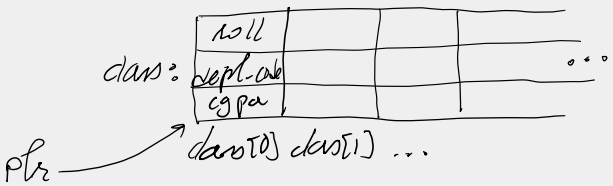


Arrow operator for struct pointers :

pBox -> width //same as (*pBox).width

Let's address some se*y questions on structure,

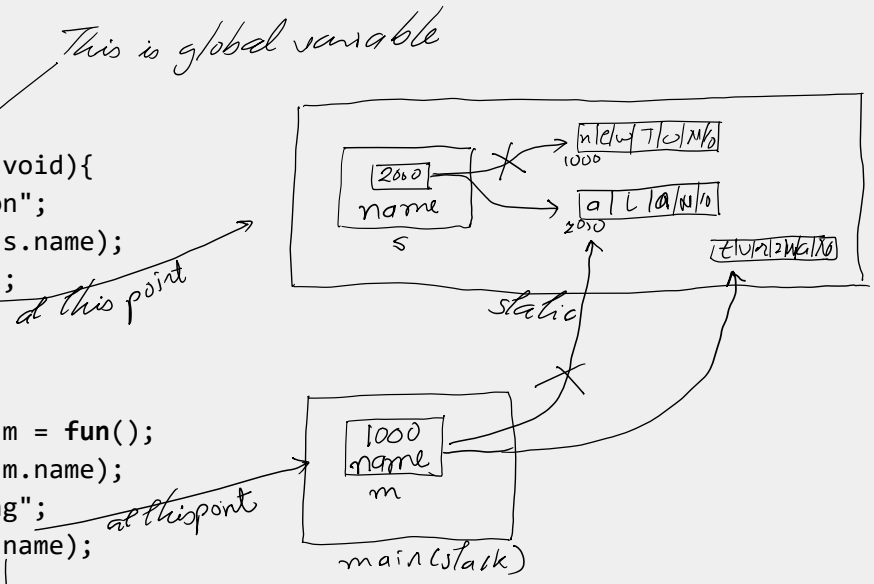
```
struct stud{
    int roll;
    char dept_code[25];
    float cgpa;
}class[10], *ptr;
...
ptr = class;
...
```



Q : Here ++ptr->roll would increment first and then points to second element of array or first get the value and then increment ? – basically this problem is of operator precedence. So, this -> have higher precedence and that is why ++ptr->roll is equivalent to ++(ptr->roll)

Q : What would be output of the following program ?

```
struct student{
    char *name;
};
struct student s;
struct student fun(void){
    s.name = "newton";
    printf("%s\n", s.name);
    s.name = "alan";
    return s;
}
void main(){
    struct student m = fun();
    printf("%s\n", m.name);
    m.name = "turing";
    printf("%s", s.name);
}
```



Output : newton alan alan

Last one is still alan because s is global and it is still pointing to alan only.

Beautiful example : Here if you print c u will get 0 because s will store NULL.

test.c > main()

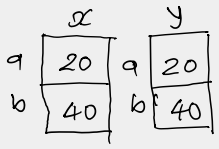
```
1  #include<stdio.h>
2  #include<string.h>
3
4  struct data{
5      struct data *s;
6      int data;
7  }a, b, c;
8  int main (){
9      a.s = &b;
10     a.data = 100;
11     b.s = &c;
12     b.data = 20;
13     printf("%d %d", a.s, b);
14     return 0;
15 }
```

inputf.txt

```
1  struct data{
2      int data;
3      struct data *s;
4  }a, b, c;
5  int main (){
6      a.s = &b;
7      a.data = 100;
8      b.s = &c;
9      b.data = 20;
10     printf("%d %d", a.s, b);
11     return 0;
12 }
```

3.3) Copying one struct to another :

```
struct PairOfInts{
    int a;
    int b;
};
struct PairOfInts x, y;
x.a = 20;
x.b = 40;
y = x;
```



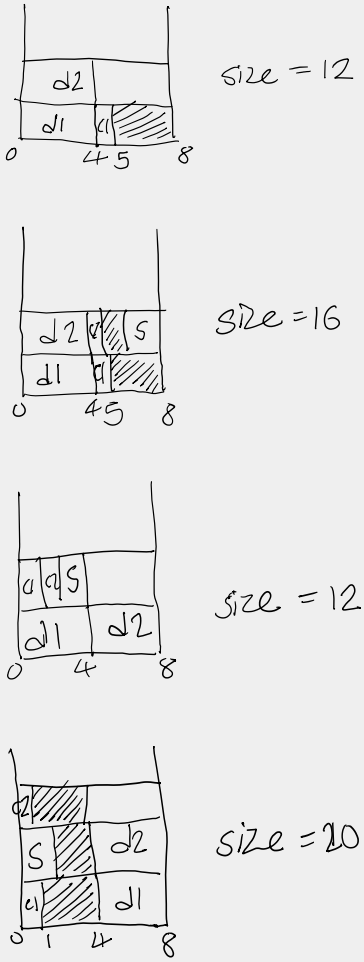
Comparing structure is not allowed in C (it will result in compile time error) but comparing elements of structure with same data type is allowed.

Size of structure and memory alignment :

Each variable of "type" must be stored at a location divisible by "sizeof(type)"

If structure contains only one type of data then size would be sizeof(one of the data)*(no. of data) if heterogeneous data types are present then due to memory alignment and padding requirements, the structure's size is influenced. To ensure efficient memory access and alignment, compilers often add padding between members to meet alignment requirements.

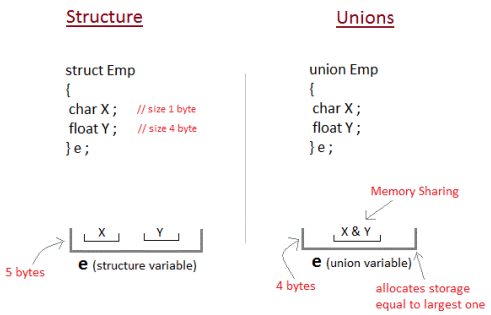
```
struct foo1 {
    int d1;
    char c1;
    int d2;
};
struct foo2 {
    int d1;
    char c1;
    int d2;
    char c2;
    short s;
};
struct foo3 {
    int d1;
    int d2;
    char c1;
    char c2;
    short s;
};
struct foo4 {
    char c1;
    int d1;
    short s;
    int d2;
    char c2;
};
```



In foo4, after the d2, there is 3 byte of padding inserted before the char c2 (in diagram it is at left which is wrong). So total size of foo4 becomes 16 + 3 (padding) + 1 (char) = 20.

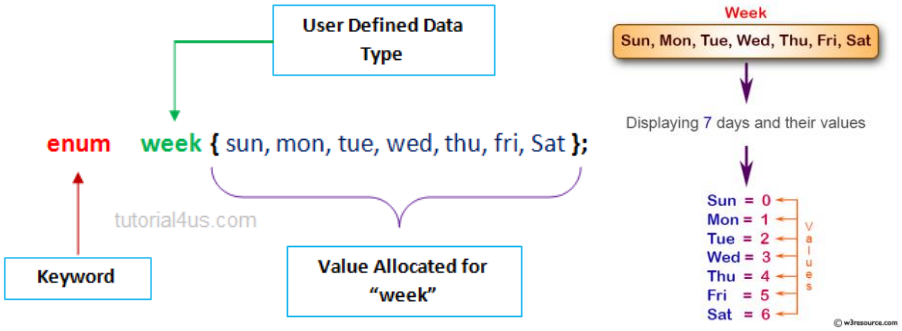
NOTE : if array is given then each element of array (not of structure) takes full max byte (if 8 byte wide structure then it will take full 8 byte).

Union : Union is same as structure but it shares memory for different data types



Here in structure 8bytes are allocated, image error.

Enum structure :



4. Miscellaneous Topics

getchar() : reading a character

scanf is overkill just to read one character

putchar() : putchar() function writes a character to screen

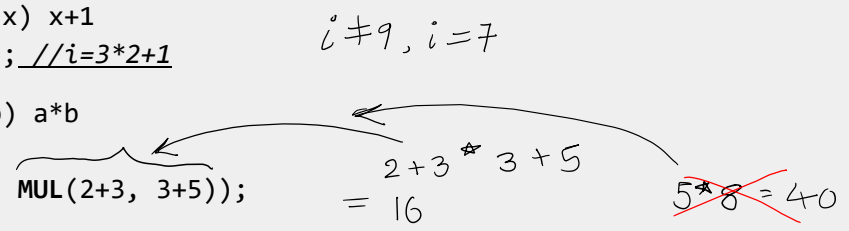
putchar(c) is equivalent to printf("%c", c);

Macros :

All lines that start with # are processed by preprocessor

```
#define plusone(x) x+1
i = 3*plusone(2); //i=3*2+1

#define MUL(a, b) a*b
int main(){
    printf("%d", MUL(2+3, 3+5));
}
```



In previous example, one thing to note that 2+3 happens at runtime because it needs CPU and we know that preprocessing happens before runtime.

In micros, Add(A, B) here A and B is called **formal parameter** or argument and passed values is called **actual parameters** or argument.

Although C does not support call-by-name parameter passing, the effect can be correctly simulated in C. We can simulate it using micros.

Call by value : just solve the expression and passed the value in some other variable so it does not affect original values.

Call by reference : It solves the expression and passes the actual parameter and affects original values.

Call by name : It does not solve the expression it passes it as it is, and affects original values.

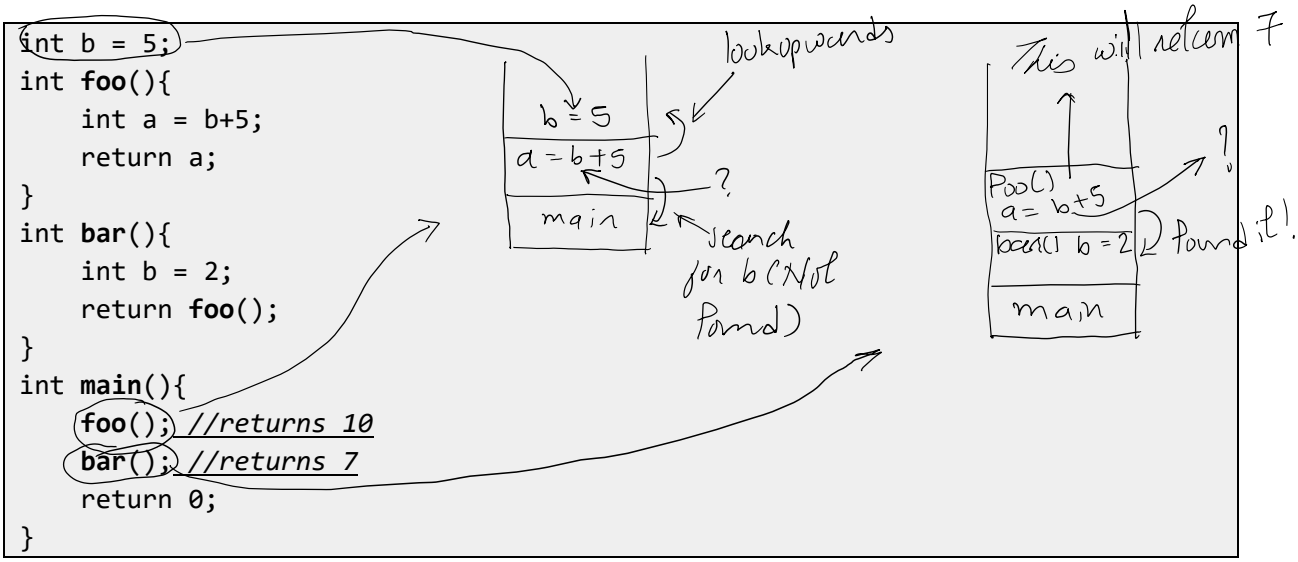
This beautiful explanation :

Static and Dynamic scoping :

Static scoping : C uses static scoping so do usual calculation

Dynamic scoping : C does not use dynamic scoping

Basic idea is when we don't know that value of variable look in stack down words and search for value of that variable if you find it then use that value if not then go up and take value. Example,



Terms related to C programming :

- **Aliases** : In computer programming, aliasing refers to the situation where the same memory location can be accessed using different names. For instance, if a function takes two pointers A and B which have the same value, then the name A aliases the name B.