Introduction to 'c' program - operator

- → Arithmetic operator (+, -, /, *, ÷)
- → Relational operator (>=, <=)(==)(!=)(<, >)
- → logical operator (&, ||, !)
- → Increment, Decrement (++, --)
- → Bitwise operator (&, ^, |, ~, >>, <<) right shift, left shift
- → Assignment operator } => =, +=, -=, /=, %=, <<=, >>=, &=, ^=, |=
- → other operator [?: & * sizeof(),]

ARITHMETIC OPERATOR

+, -, *, /, %.

→ all are (binary operator) → they need "2" operands.

Operator precedence and (Associativity) → 2 or more operators are of same expression

Highest	*, /, %	- left to right
Lower	+, -	- left to right

Increment and Decrement → (Unary operator)

Increment

Increment the value of a variable by one.

```
int a=5
a++;
```

a=6

Decrement

Decrement the value of a variable by one

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

a=4

a = 5

a++; => a = a + 1 = 5 + 1 = 6

a--; => a = a - 1 = 5 - 1 = 4

2 types < Pre / Post

→ Pre-increment

++a;

x = ++a

x: 6 = a: 6

Post-increment

a++; x a

5 = 5 6

→ Pre decrement

--a;

→ Post decrement

a--;

Ivalue → left value → object that has an identifiable location in memory. (i.e having an address).

must be a variable. } because they have the capability to store the data.

Ivalue - can't be a function.

rvalue → no identifiable location in memory

→ anything which is capable of returning a constant expression or value.

→ its may be a (expression)

→ its not a variable.

Token generation

→ lexical analysis is the first phase in compilation process

↓

like a scanner

↓

its scanns the source program to meaningful sequence of character

↓

token

Token

↓

mapped into token - name and attribute - value

example => int → < keyword, int >

=> lexical → obj always matches the longest character sequence

(X) Post increment / decrement in context of equation

a=4
b=3

a+++b

a++ +b

=> first use value in the equ.

then → increment the value

4 + 3 = 7

then a = 5

(X) Pre increment / decrement in context of equation

=> first increment the value and then use in the equation after completion of the equation

a = 4 b = 3

a + ++b

a + 4

4 + 4 = 8

HW a = 4, b = 3

a+ + + + + b

4 + 4 = 8

Relational operator → will return either true / false

equal to ==

not equal to !=

less than or equal to <=

Greater than or equal >=

less than <

Greater than >

Logical operator

&&, ||, !

and. or logic not

AND, OR → used to combine 2 condition

&& - returns true → all the condition want to be true.

→ returns false → one or more than condition is false

|| → returns → true → one or more the one condition will be true.

→ returns false → one or all conditions are false

! - returns true → when condition is false

" false - when all condition is true

Short circuit → condition anywhere in the expression that returns false, then the rest of the condition after all with will not evaluated.

Bitwise operator

AND	&
OR	|
NOT	~
left shift	<<
right shift	>>
xor	^

7 = 0111
4 = 0100
4 = 0100

7 = 0111
4 = 0100
7 = 111

binary operator

left shift << → 2 operand needed.

① → when bits → shifted lefted - then trailing pos are filled with zero

ex

Var << 1 Var = 3

0000 0011 → 3
0000 0110 → 6

② Left shifting is equivalent to multiplication by 2nd right operand

Var = 3 → 0000 0011

Var << 1 → 0000 0110

→ 3 x 2¹

Right shift >> → binary operator

① bit are shifted right, leading pos are filled with "0"

Var = 3

Var >> 1

0000 0011 ⇒ 3

0000 0001 = 1

leading pos → 0

(2) Right shift is = division by 2 right operand

var = 3

var >> 1 → $\left(\frac{3}{2^1}\right)$

Bitwise - XOR

X-OR

A	B	C
0	0	0
0	1	1
1	0	1
1	1	0

7 - 0 1 1 1

^

4 - 0 1 0 0

3 0 0 1 1

7 ^ 4 = 3

Scope = life time → Variable is declared, defined & used and when a block / region ends Variable automatically destroyed.

Variable

local
↓
(within a region)

global
↓
available for whole program.
default initialized to "0"

Variable modifier :- Auto & extern

Auto modifier. auto int some_variable_name

↳ it have some garbage vale at initial stage

EXTern modifier (auto int var) - syntax

↳ simply means → declaration
→ no memory space

Syntax : (extern int a)

Modified reg:

→ reg keyword

↳ hints the compiler to store a variable in reg memory