VARIABLES AND CONSTANTS

VARIABLES	CONSTANTS
1. A **variable** is an entity whose value can vary (i.e. change) during the execution of a program.	1. A **constant** is an entity whose value remains same throughout the execution of a program.
2. The value of a variable can be changed because it has a modifiable l-value. Since, it has modifiable l-value, it can be placed on the left side of assignment operator.	2. It cannot be placed on the left-side of assignment operator because it does not have a modifiable l-value.
3. Variable can also be placed on the right side of assignment operator. Hence, it has r-value too. Thus, a variable has both l-value and r-value.	3. It can only be placed on the right side of assignment operator. Thus, a constant has an r-value only

---

CONSTANTS

- **Literal constant** or just **literal** denotes a fixed value, which may be an integer, floating point number, character or a string. The type of literal constant is determined by its value.
- **Symbolic constants** are created with the help of define preprocessor directive .

  For example: #define PI 3.14124 defines PI as a symbolic constant with value 3.14124. Each symbolic constant is replaced by its actual value during the preprocessing stage .
- **Qualified constants** are created by using const qualifier. The following statement creates a qualified character constant named a:

```
const char a='A';
```

Since, qualified constants are placed in the memory, they have l-value. But, as it is not possible to modify them, this means that they do not have modifiable l-value i.e. they have non-modifiable l-value. E.g.

```
int a=10;                 //  It is possible to modify the value of a.
const int a=10;.          //it is possible read the value placed within the memory location,
but it is
                              not possible to modify  the value.
```

***Integer constants*:-** (Rules/properties)

(a) It must have at least one digit

(b) not have decimal point

(c) can be +ve or -ve, but by default(if not defined) +ve

(d) No commas/blanks are allowed within integer constant.

(e) Range → -32768 to 32767 (for 16-bit compiler)

(f) Integer constant:023=>23 in octal no.sys;0x23=> 23 in hex no.sys;23L or 23l=>long int, 23U or 23u=>unsigned int

Real Constants **:-** (also called as Floating point Const.)

Can be written as –: fractional form Or exponential form

(a) It must have atleast one digit.

(b) Decimal point present.

(c) +ve or –ve, default +ve

(d) No comma/blank allowed.

(e) It is represented in exponential form if the value is either too small or too large.

In exponential form two parts of constant →

Mantissa	e	exponent

Range : -3.4 e 38 to 3.4 e 38 eg: +3.2 e^{-5} or 4.1 e8

***Character Const*:-**

Single – alphabet/digit/special symbol kept within single inverted commas.

e.g. 'A', 'I', '5', '='

List of Escape Sequences

ESCAPE SEQUENCES	CHARACTER VALUE	ACTION ON OUTPUT DEVICE
'\''	Single quotation mark	Prints '
'\"'	Double quotation mark (")	Prints "
'\?'	Question mark (?)	Prints ?
'\\'	Backslash character (\)	Prints \
'\a'	Alert	Alerts by generating beep
'\b'	Backspace	Moves the cursor one position to the left of its current position.
'\f'	Form feed	Moves the cursor to the beginning of next page.
'\n'	New line	Moves the cursor to the beginning of the next line.
'\r'	Carriage return	Moves the cursor to the beginning of the current line.
'\t'	Horizontal tab	Moves the cursor to the next horizontal tab stop.
'\v'	Vertical tab	Moves the cursor to the next vertical tab stop.
'\0'	Null character	Prints nothing

Expression: It is made up of one or more operands and operators in it. as: a=b+c

Primary expression – It consists of only one operand with no operator.

eg.

Name,	literal constant,	parenthetical expression
a, b12,	5,123.98,	
price,	'A', "welcome"	

INT_MAX

(2 * 3+4), (a = 23 + b * 6) ←

Simple expression: a+b has 1 operator: '+'
Compound expression: c=a+b has 2 operators: '+' & '='
Thus, an expression is a sequence of operands and operators that specifies the computation of a value.

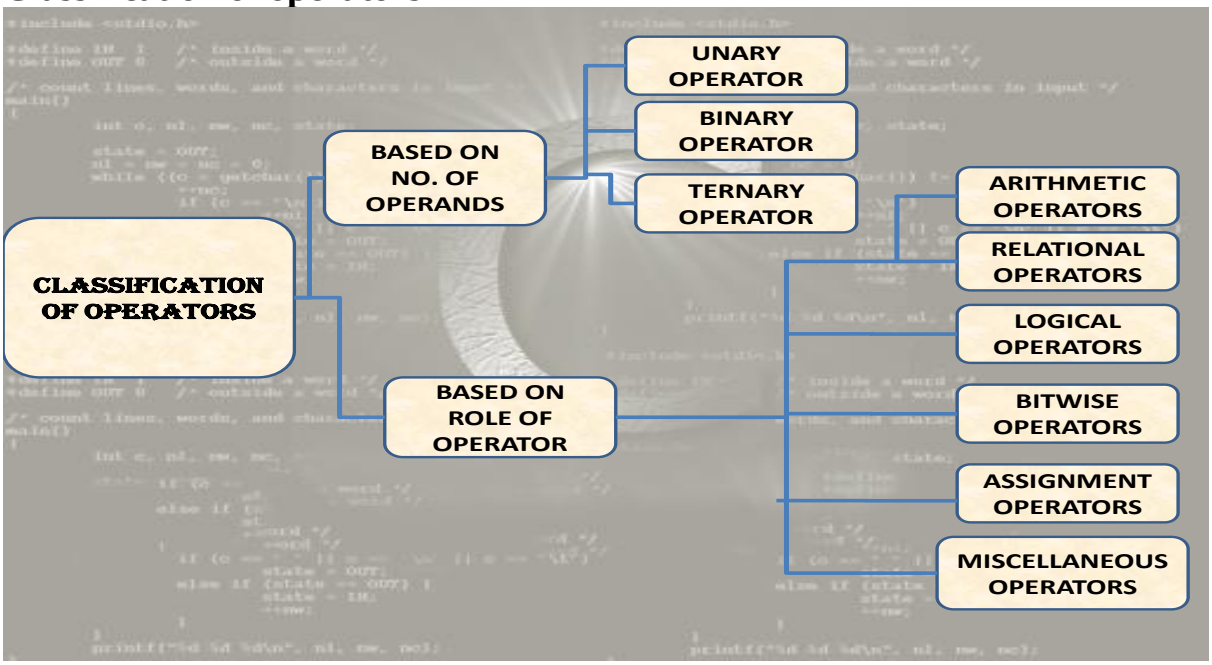
Assignment expression(use of assignment operator:'='):
It evaluates the operand on the right side of the operator (=) & keeps its value in the variable on the left.
2 types of assignment expressions.

Simple Assignment	Compound Assignment
a = 5,	x * = y +3 ≡ {x = x*(y+3)},
b = x +1,	
i = i+1	x - = 4 ≡ {x = x-4}

Operands: it is an entity on which an operation is to be performed. It can be : a var, const, function call or a macro name. Eg. In expression:a+b, a & b are operands.

Operators: It specifies an operation to be performed on its operands.
Eg. In expression:a+b, '+' is an addition operator applied on operands :a & b.

Classification of operators



Types of operators,(based on no. of operands used):

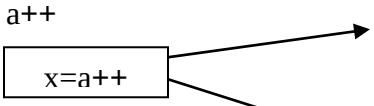
1.Unary operators

It operates on one operand (operand on left *or* on right of operator),
as: in expression "-3", '-' is a unary minus operator It operates on 3 only.
Other unary operators are : ++(increment operator), --(decrement operator), &(address of operator), *sizeof( )* operator,!(logical not operator), ~(bitwise not operator) etc.
Expression types on the basis of position of ++ and --(Unary operators):
PostFix & PreFix expression : ++, - -

1. prefix: operator , operand, ex: ++ a(increment a by 1=>(a= a+1))

2. Post fix: operand, operator Variable, ex: a++(increment a by 1=>(a= a+1))

Evaluation of postfix ++: (first use then increment)



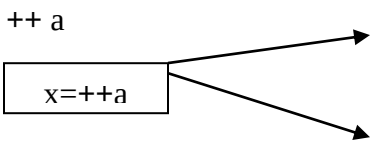
x = a

step(1) value of expression is a(assigned to x)

a = a +1

step(2) value of a is incremented by 1(assigned to a)

Evaluation of prefix: (first increment then use)



a = a +1

(1) value of a is incremented by 1(assigned to a)

x = a

(2) value of expression is a after increment(assigned to x).

Similarly for a-- & --a

Both are: a= a – 1, after execution.

Examples to use above property: Difference between x++ and ++y

```
main()
{
int a,b,x=10,y=10;
a = x++;
b = ++y;
printf("Value of a : %d%d",a,x++);
printf("\nValue of b : %d%d",b,++y);
}
```

```
O/P:
```

```
#include <stdio.h>
Void  main(){
  int c=2,d=2;
   printf("%d\n",c--); //this statement displays 2 then, only c incremented by 1 to 3.
   printf("%d",++c);   //this statement increments 1 to c then, only c is displayed. }
```

Size of () :

It tells us the size, in bytes of a type or a primary expression.

Eg.

sizeof (int); => 2, size of integer data type is 2 bytes or 4 bytes.

sizeof(-345.23); => 4, size of floating point data type is 4 bytes

sizeof (x)

2.Binary operators

It operates on two operands (one operand on left and other on right of operator),

It is left to right associative

as: in expression “10+4”, ‘+’ operates on 10 and 4.

Other binary operators are: +, -, /, %, <<(left shift operator),==(equality operator), &&(logical and operator) etc.

3.Ternary Operator (conditional operator)

It operates on **three operands**. As: conditional operator(**? :**) is ternary operator.

Ex:

```
#include <stdio.h>
 main(){
     int d1,d2, larger;
   printf("Enter two numbers a & b: ");
   scanf("%d%d",&d1,&d2);
   larger=d1>d2?d1:d2;
  printf("Larger Number is = %d",larger);
}
```

- The conditional operator operator (**?:**) is just like an if .. else statement that can be written within expressions.
- The syntax of the conditional operator is

  exp1 ? exp2 : exp3

Here, exp1 is evaluated first. If it is true then exp2 is evaluated and becomes the result of the expression, otherwise exp3 is evaluated and becomes the result of the expression. For example,

large = (a > b) ? a : b;

- Conditional operators make the program code more compact, more readable, and safer to use as it is easier both to check and guarantee that the arguments that are used for evaluation.
- Conditional operator is also known as ternary operator as it is neither a unary nor a binary operator; it takes *three* operands.

Example:

main() { int a=2,b=3,c; **c = (a > b) ? a : b;** printf("%d",c); }	*main()* *{* *int a=2,b=23,c=1, biggest ;* ***biggest = a > b ? ((a > c ? a : c) : (b > c ? b : c)) ;*** *printf("\nThe biggest number is : %d", biggest) ;* *}*
O/P: 3	Output: 23

- **Any statement written as a comment will never be executed by complier.**
- **No nesting is possible in comments.**

```
/*- - -- --- -
        /*- - -- --- -
                - - - - -- */                    invalid
                - - - - - --
                - - - - -- */
```

Arithmetical Operators: addition(+),subtraction(-),Multiplication(*),division(/),modulus(%) etc.

Modulus operator (%) :

It returns the remainder as a result after division operation

4 %3 = 1, -5 % 2 = -1

7% 3 = 1, 5 % -2 = 1, 2 % 3 = 2

Logical data :- A piece of data is called logical if it conveys the idea of true or false.

C has *no logical data type.* Integer can be used to represent logical data.

If a data Value is zero('0') → false

If a data Value is non-zero (+ve or -ve)→ true

Logical operators:- 3 logical operators are defined in C-

'Logical not'(!), 'Logical and'(&&), 'Logical or'(||).

- They are used for combining logical values & designing new logical values.

'not' operator (!) :

- It is a unary operator

.It changes a true value $\xrightarrow{to}$ false and a false value $\xrightarrow{to}$ true

'and' operator (&&):

- It is a binary operator
- 4 combinations are possible in its operands.
- Result is → true only if both operands → true.
  Otherwise always → false.

'or' operator (| |):

- It is a binary operator
- 4 combinations are possible in its operands.
- The result is false if both operands → false
  Otherwise always → true.

Logical operators can be shown in a Table called as – truth- table:

not			and			or		
x	!x		x	y	x&&y	x	y	x || y
f	t		f	f	f	f	f	f
t	f		f	t	f	f	t	t
			t	f	f	t	f	t
			t	t	t	t	t	t

t: true & f:false.

Expressions connected by && or || are evaluated **left to right** and the evaluation stops as soon as truthfulness or falsehood of the result is known. Thus, in an expression:

a. E1&&E2, where E1 and E2 are sub-expressions, E1 is evaluated first. If E1 evaluates to 0 (i.e. false), E2 will not be evaluated and the result of overall expression will be 0 (i.e. false). If E1 evaluates to a non-zero value (i.e. true) then E2 will be evaluated to determine the value of overall expression.

b. E1||E2, where E1 and E2 are sub-expressions, E1 is evaluated first. If E1 evaluates to a non-zero value (i.e. true), E2 will not be evaluated and the result of overall expression will be 1 (i.e. true).

If E1 evaluates to 0 (i.e. false) then E2 will be evaluated to determine the truth value of overall expression.

Example:

void main() { **int a=10,b=-5;** **int x=3,y=0;** **int c,d,u,v;** **c=a&&b;** **d=a||b;** **u=x&&y;** **v=x||y;** **printf("%d%d%d%d",c,d,u,v);** }	**main()** **{** **int i=0,j=0,k=2,l;** **l=i||j&&k;** **printf("%d%d%d%d",i,j,k,l);** **}**
O/P-> **1 1 0 1**	**O/P:0 0 2 0**

Comparative Operators: '6' comparative operators are provided by C.

we can keep them in to two groups – **relational operator & Equality operator**.

KEY POINTS:RELATIONAL OPERATORS

- ✓ There should be no white space character between two symbols of a relational operator.
- ✓ The result of evaluation of a relational expression (i.e. involving relational operator) is a boolean constant i.e. 0 or 1.
- ✓ Each of the relational operators yields 1 if the specified relation is true and 0 if it is false. The result has type int.
- ✓ The expression a<b<c is valid and is not interpreted as in ordinary mathematics. Since, less than operator (i.e. <) is left-to-right associative, the expression is interpreted as (a<b)<c. This means that "if a is less than b, compare 1 with c, otherwise, compare 0 with c".
- ✓ An expression that involves a relational operator forms a **condition**. For example, a<b is a condition.

Ex:

main() **{** **int a=2,b=3,c=1,d;** **d=a<b;** **printf("%d",d);** **}**	**main()** **{** **int a=2,b=3,c=1,d;** **d=a<b>c; //it is:(a<b)>c** **printf("%d",d);** **}**
Output: 1	**Output: 0**

OPERATOR	MEANING
==	RETURNS 1 IF BOTH OPERANDS ARE EQUAL, 0 OTHERWISE
!=	RETURNS 1 IF OPERANDS DO NOT HAVE THE SAME VALUE, 0 OTHERWISE

Relational operator complements

> $\xrightarrow{Complement\text{-}of}$ <=

< $\xrightarrow{Complement\text{-}of}$ >=

= = $\xrightarrow{Complement\text{-}of}$!=

We can also simplify operators as:

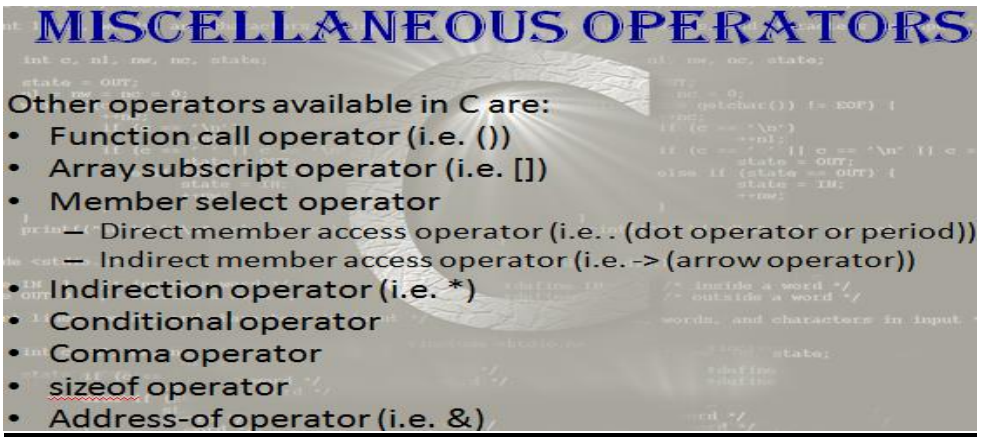
!(x >y) ↔ x < = y

! (x < y) ↔ x > = y

!(x! = y) ↔ x = = y

Programs objectives to utilize Relational & Logical Operators:

1. Wap to check whether i/p no. is even or odd.
2. Wap to check if the person is eligible for voting or not.
3. Wap to check if the i/p year is Leap year or not
4. Wap to calculate the salary of a person after increment, given conditions are- if experience is between 0 to 3 years then 10% increment otherwise 20% increment will be given.



Precedence and associativity Of Operators

The order in which the operators will operate depends upon the precedence and associativity.

Associativity of operators:

When an expression has two operators of equal priority/precedence the tie between them is settled using the associativity of the operators.

Associativity types: → left to right
→ right to left

eg: **a = 3/2 * 5**

here '/' & * have same priority & both also use left to right associativity.

So '/' or * which one should perform its operation first.

so '/' will be performed first.

'=' → has R to L associativity so in "a = b = 3", b = 3 will be operated first.

a = b =c= 3 => a=(b=(c=3))

In arithmetic operators precedence is same as Hierarchy among operators:

1st priority → Parenthesized value(value within bracket or paranthesis): a*(b/(c-d))

2nd priority → *, /, %.

3rd priority → +, -

4th priority → =

Direction of solving (associativity) is from left to right in an expression.

i = 2 * -3 / 4 + 4/4 + 8 -2 + 5%8

Result →

6/4 + 4/4 +8 - 2 + 5/8

1 + 4/4 + 8 - 2 + 5/8

1 + 1 + 8 -2 + 5/8

10 – 2 + 0

8 + 0 → 8

A=2+5%7-1*6{Hint: apply brackets to solve as per precedence}

B= 6*-3%-9+18/3-5

Calculate the results of following expression by applying precedence & associativity rules:

1. **a-b/(3+c)*(2-1); a=9,b=12,c=3.** Ans:
2. **a-(b/(3+c)*2)-1; a=9,b=12,c=3.** Ans:4
3. **4+5-55/5%-10 .** Ans:8

Precedence of Operators

•Each operator in C has a **precedence** associated with it.
•In a compound expression, if the operators involved are of different precedence, the operator of higher precedence is evaluated first.

•For example, in an expression b=2+3*5,

•The sub-expression 3*5 involving multiplication operator (i.e. *).

•The result of evaluation of an expression is an r-value. The sub-expression 3*5 evaluates to an r-value 15. This r-value will act as second operand for addition operator and the expression becomes b=2+15.

Associativity of operators:

When an expression has two operators of equal priority/precedence t
between them is settled using the associativity of the operators.

Associativity types: → left to right
→ right to left

eg:
a = 3/2 * 5
here '/' & * have same priority & both also use left to right associativity.
So '/' or * which one should perform its operation first.
so '/' will be performed first.
'=' → has R to L associativity so in "a = b = 3", b = 3 will be operated first.
a = b =c= 3 => a=(b=(c=3))

a=2+5%7-1*6

2+(5%7)-(1*6)=2+(5)-(6)=2+5-6=7-6=1

A= 6*3%9+18/3-5

After solving above A=1.

Table: Name, Symbol, Category, Precedence and Associativity of Operators

S.No	Operator	Name of Operator	Category	-ary of Operators	Precedence	Associativity
1.	() [] -> .	Function call Array subscript Indirect member access Direct member access			Level-I (Highest)	
2.	! ~ + - ++ -- & * sizeof	Logical NOT Bitwise NOT Unary plus Unary minus Increment Decrement Address-of Deference Sizeof	Unary	Unary	Level-II	R→L
3.	* / %	Multiplication Division Modulus	Multiplicative operators	Binary	Level-III	L→R
4.	+ -	Addition Subtraction	Additive operators	Binary	Level-IV	L→R
5.	<< >>	Left Shift Right Shift	Shift operators	Binary	Level-V	L→R
6.	< > <= >=	Less than Greater than Less than or equal to Greater than or equal to	Relational operators	Binary	Level-VI	L→R
7.	== !=	Equal to Not equal to	Equality operators	Binary	Level-VII	L→R
8.	&	Bitwise AND	Bitwise operator	Binary	Level-VIII	L→R
9.	^	Bitwise X-OR	Bitwise operator	Binary	Level-IX	L→R
10.	|	Bitwise OR	Bitwise operator	Binary	Level-X	L→R
11.	&&	Logical AND	Logical operator	Binary	Level-XI	L→R
12.	||	Logical OR	Logical operator	Binary	Level-XII	L→R
13.	?:	Conditional operator	Conditional	Ternary	Level-XIII	R→L
14.	= *= /= %= += -= &= |= ^= <<= >>=	Simple assignment Assign product Assign quotient Assign modulus Assign sum Assign difference Assign bitwise AND Assign bitwise OR Assign bitwise XOR Assign left shift Assign right shift	Assignment & Shorthand assignn operators	Binary	Level-XIV	R→L
15.	,	Comma operator	Comma	Binary	Level-XV (Least)	L→R

Data-Type Conversion

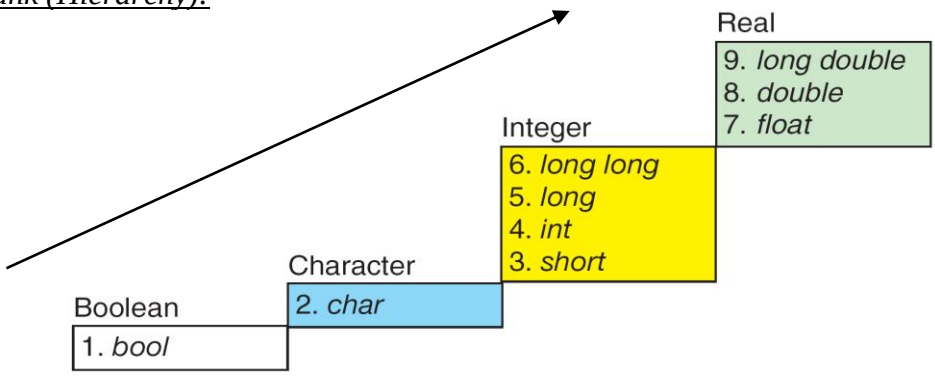
In any expression if two variables of different data-types are used, then at least one of them will be converted into another, and then only that operation can be performed.
Two Categories of type conversion –
(A) Implicit
(B) Explicit(type-casting)

(A) Implicit Type Conversion :- ***When the 2 operands of different types are used in a binary expression, c compiler automatically converts one type to another, this is called implicit type conversion.***

- This is generally done with the intermediate values during evaluation of expression.

Rules to perform implicit type conversion are as based on hierarchy:

Conversion Rank (Hierarchy):



- In assignment expression :

L.H.S. = R.H.S.; then there may be 2 kinds of conversion exists-

Promotion – if the right expression (R.H.S.) has lower rank so it is promoted.
Demotion – if the right expression (R.H.S.) has a higher rank so it is demoted. Demotion occurs only in assignment operation implicitly.

Since small object can be easily kept in a big box so **promotion** is always possible but reverse is not true.

eg:

```
char c = 'A';
int i = 1234;
long double d = 3458.0004;
```

now **promotion occurs in following independent assignments:**

```
i = c;          - value of i = 65
d = i;          - value of d is 1234.0000
```

- So in **Demotion** – If the size of the variable at the left side can accommodate the value of the expression, it is O.K. otherwise results may be unpredictable.

eg.:
```
short s = 78;
char c = 'A';          int j = INT_MAX(=32767);
int k = 65;
s = j          - value of s may be - unexpected
c = k+1        - demotion : value of c is 'B'.
```

Conversion in other Binary Expressions: Summary of rules can be followed in these 3 steps:

1. The operand with the higher rank is determined using the ranking in hierarchy figure
2. The lower-ranked operand is promoted to the rank defined in 1st step. In 2nd step after promotion, both expression have same rank.
3. The operation is performed with the expression value having the type of promoted rank.

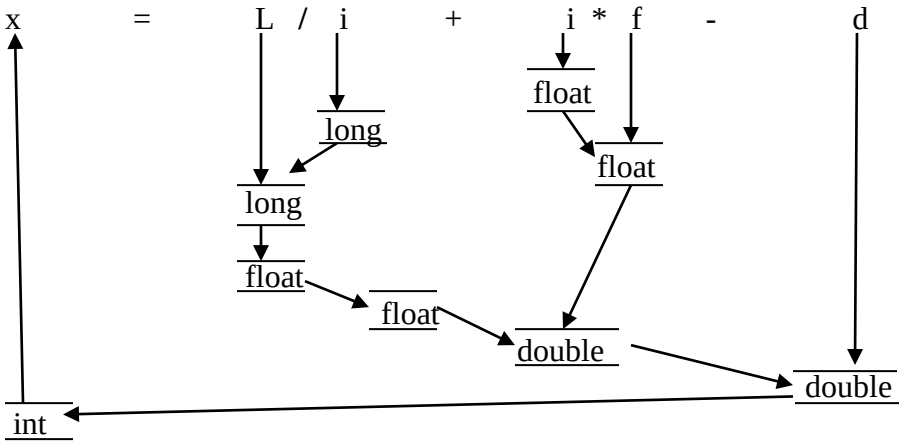
ex:
```
char   c = 'A';        int      i = 3650;      short   s = 78;
long double d = 3458. 0004;
i*s;    // result is an int
d*c;    // result is long double.
```

In case of demotion type-conversion is done as –

- Float to int causes truncation of the fractional part.
- Double to float causes rounding of digits.
- Long int to int causes dropping of the excess higher order bits.

eg:
```
int i, x;
float f;
double d;
```

long int L;



Integer & Float conversions:

(a) Integer {(arithmetic operation)} integer = Integer (result)
(b) Real (float) {(arithmetic opn.)} real (float) = Real (result)
(c) Integer {(arithmetic opn.)} real (float) = Real (float) result.
5.0/2 = 2.50000
5/2 = 2
5/2.0 = 2.5

Explicit conversion :-

- By this we can convert data from one type to another forcefully (not automatically)
- It is done by using a unary operator – cast operator: "( )" and this operation is called-TypeCasting
- To cast (convert) data from one type to another, we specify data-type (new) in parenthesis before the value we want to convert.

eg.: to convert an integer, a , to a float : (float) a

- Like any other operation, the value of a is still of type int, but the value of the expression is promoted to float.

General form of explicit conversion is :

(data type-name) expression

ex:

a= (int)21.3/(int)4.5
so result stored in a is 5
y = (int)(a+b), **The result of a+b is converted to int.**

(float) (a/10) + (float) a/10

(if a=3) 0.0 0.3

```
int a=3;
printf("%f\n",float(a/10));   =>0.0000
printf("%f\n",(float)a/10);  =>0.3000
```

x= (int) (y+0.5)

If y = 27.6 → y +0.5 = 28.1 but after casting x = 28 is stored. So, the result is changed not the expression

- It is always true that we should use explicit conversion (casting), instead of leaving it on system.

```
main()
{
    float u=3.5;
    int v,w,x,y;
    v=(int)(u+0.5);
```

```
main()
{
    int u=3.5,v,w,x,y;
    v=(int)(u+0.5);
    w=(int)u+0.5;
```

w=(int)u+0.5; x=(int)((int)u+0.5); y=(u+(int)0.5); printf("%d%d%d%d",v,w,x,y); }	x=(int)((int)u+0.5); y=(u+(int)0.5); printf("%d%d%d%d",v,w,x,y); }
Output:4333	3333
main() { char aa='c'; int b=97; printf("%d",aa+10); printf("\t%c",b-10); }	main() { int i=5; printf("%d%d%d%d%d",i++,i--,++i,--i,i); }
Output:109 w	45555

Bit-wise operators

S.No	Operator	Name of Operator	Category	ary of Operators	Precedence amongst bitwise class	Associativity
1.	~	Bitwise NOT	Unary	Unary	Level-I	R→L
2.	<< >>	Left Shift Right Shift	Shift operators	Binary	Level-II	L→R
3.	&	Bitwise AND	Bitwise operator	Binary	Level-III	L→R
4.	^	Bitwise X-OR	Bitwise operator	Binary	Level-IV	L→R
5.	|	Bitwise OR	Bitwise operator	Binary	Level-V	L→R

Usage of Bitwise Operations or Why to Study Bits

1. **Compression:** Occasionally, you may want to implement a large number of Boolean variables, without using a lot of space. A 32-bit int can be used to store 32 Boolean variables. Normally, the minimum size for one Boolean variable is one byte. All types in C must have sizes that are multiples of bytes. However, only one bit is necessary to represent a Boolean value.

2. **Set operations:** You can also use bits to represent elements of a (small) set. If a bit is 1, then element is in the set, otherwise it's not. You can use bitwise AND to implement set intersection, bitwise OR to implement set union.

3. **Encryption:** swapping the bits of a string for e.g. according to a predefined shared key will create an encrypted string.

Properties of Bitwise operators:

- ✓ Bitwise operators operate on the individual bits of operands and are used for bit manipulation.

- ✓ They can only be applied on operands of type char, short, int, long, whether signed or unsigned (never on float/double).

- ✓ The bitwise-AND and bitwise-OR operators operate on the individual bits of the operands according to the truth tables.

- ✓ EX:The expression 2&3 evaluates to 2 and 2|3 evaluates to 3.

Bitwise AND

In C, the & operator is bitwise AND. The following is a chart that defines &, defining AND on individual bits.

x_i	y_i	x_i & y_i
0	0	0
0	1	0
1	0	0
1	1	1

We can do an example of bitwise &. It's easiest to do this on 4 bit numbers, however.

Variable	b_3	b_2	b_1	b_0
x	1	1	0	0
y	1	0	1	0
z = x & y	1	0	0	0

Int x=12, y=10 then, printf("%d",x&y) will give 8 in decimal. printf("%d",x|y) will give 14 in decimal. printf("%d",x^y) will give 6.

Bitwise OR

The | operator is bitwise OR (it's a single vertical bar). The following is a chart that defines |, defining OR on individual bits.

x_i	y_i	$x_i \mid y_i$
0	0	0
0	1	1
1	0	1
1	1	1

We can do an example of bitwise |. It's easiest to do this on 4 bit numbers, however.

Variable	b_3	b_2	b_1	b_0
x	1	1	0	0
y	1	0	1	0
z = x | y	1	1	1	0

Bitwise XOR

The ^ operator is bitwise XOR. The usual bitwise OR operator is *inclusive* OR. XOR is true only if exactly one of the two bits is true.

The following is a chart that defines ^, defining XOR on individual bits.

x_i	y_i	$x_i \wedge y_i$
0	0	0
0	1	1
1	0	1
1	1	0

We can do an example of bitwise ^. It's easiest to do this on 4 bit numbers, however.

Variable	b_3	b_2	b_1	b_0
x	1	1	0	0
y	1	0	1	0
z = x ^ y	0	1	1	0

Bitwise NOT

There's only one unary bitwise operator, and that's bitwise NOT. Bitwise NOT flips all of the bits.

The following is a chart that defines ~, defining NOT on an individual bit.

x_i	$\sim x_i$
0	1
1	0

We can do an example of bitwise ~. It's easiest to do this on 4 bit numbers (although only 2 bits are necessary to show the concept).

Variable	b_3	b_2	b_1	b_0
x	1	1	0	0
z = ~x	0	0	1	1

Left shift Operator "<<"

The left shift operator will shift the bits of x(decimal no. represented in binary digits) towards left for the given number of times(n) as x<<n

```
int a=2<<1;
```

Let's take the binary rep

resentation of 2 assuming int is 1 byte for simplicity.

```
Position 7    6    5    4    3    2    1    0
Bits     0    0    0    0    0    0    1    0
```

Now shifting the bits towards left for 1 time, will give the following result

```
Position 7    6    5    4    3    2    1    0
Bits     0    0    0    0    0    1    0    0
```

Now the result in decimal is 4. You can also note that, 0 is added as padding in the position 0.

If you left shift like 2<<2, then it will give the result as 8. **Therefore left shifting n times, is equal to multiplying the value (x) by 2^n.**

Right shift Operator ">>"

The right shift operator will shift the bits towards right for the given number of times.

```
int a=8>>1;
```

Let's take the binary representation of 8 assuming int is 1 byte for simplicity.

```
Position    7    6    5    4    3    2    1    0
Bits        0    0    0    0    1    0    0    0
```

Now shifting the bits towards right for 1 time, will give the following result

```
Position 7    6    5    4    3    2    1    0
Bits     0    0    0    0    0    1    0    0
```

Now the result in decimal is 4. Right shifting 1 time, is equivalent to dividing the value by 2. You can also note that, 0 is added as padding in the position 7.

Therefore right shifting n times, is equal to dividing the value (x) by 2^n.

What does the following code do?

1. int x = 3 ;
2. int n = 2 ;
3. x << n ;
4. printf("%d\n", x) ; O/P will be:3 or 12; ans is 3.

To get answer as 12 change line no. 3 and 4 as:

x = x << n ;

printf("%d\n", x) ;

Note on shifting signed and unsigned numbers

While performing shifting, if the operand is a signed value, then **arithmetic shift** will be used. If the type is unsigned, then **logical shift** will be used.

In case of arithmetic shift, the sign-bit (MSB) is preserved. Logical shift will not preserve the signed bit. Let's see this via an example.

```
#include<stdio.h>
int main() {
   signed char a=-8;
   signed char b= a >> 1;
   printf("%d\n",b);
}
```

In the above code, we are right shifting -8 by 1. The result will be "-4". Here arithmetic shift is applied since the operand is a signed value.

```
int main() {
   unsigned char a=-8;
   unsigned char b= a >> 1;
```

```
    printf("%d\n",b);
}
```

Note**: Negative number are represented using 2's complement of its positive equivalent.**
2's compliment of +8 is: 1111 1000

Right shifting by 1 yields, 0111 1100 (124 in decimal)

The Magic of XOR(+) or ^

Properties of XOR

Here are several useful properties of XOR. This applies to plain XOR and bitwise XOR.

- **x (+) 0 = x**

XORing with 0 gives you back the same number. Thus, 0 is the identity for XOR.

- **x (+) 1 = ~x**

XORing with 1 gives you back the negation of the bit. Again, this comes from the truth table. **For bitwise XOR, the property is slightly different: x ^ ~0 = ~x.**
That is, if you XOR with all 1's, the result will be the bitwise negation of x.

- x (+) x = 0

XORing x with itself gives you 0. That's because x is either 0 or 1, and 0 (+) 0= 0and 1 (+) 1 = 0.

- **XOR is associative.**

That is: (x (+) y) (+) z = x (+) (y (+) z).
You can verify this by using truth tables.

- **XOR is commutative.**

That is: x (+) y = y (+) x.
You can verify this by using truth tables.

Swapping without "temp"

```
temp = x ;
x = y ;
y = temp ;
```

Now solve this without using a temp variable. This means you can ONLY use x and y. This does NOT mean that you name the variable temp2.

```
x = x ^ y ;
y = x ^ y ;
x = x ^ y ;
```

The key to convincing yourself this works is to keep track of the original value of x and y. Let A be the original value of x(that is, the value x has just before running these three lines of code).
Similarly, let B be the original value of y.
We can comment each line of code to see what's happening.

```
// x == A, y == B
x = x ^ y ;
// x == A ^ B, y == B
y = x ^ y ;
// x == A ^ B
// y == (A ^ B) ^ B == A ^ (B ^ B) (by Assoc)
// == A ^ 0 (by z ^ z == 0 property)
```

```
// == A (by z ^ 0 == z property)
x = x ^ y ;
// x == ( A ^ B ) ^ A
// == ( A ^ A ) ^ B (by Assoc/Commutativity)
// == 0 ^ B (by z ^ z == 0 property)
// == B (by z ^ 0 == z property)
// y == A
```

After the second statement has executed, y = A. After the third statement, x = B.