Relational Operators: Relational Operators are symbols that are used to test the relationship between 2 variables or between variable and a constant. We often compare two quantities & depending upon this relation take certain decisions.

Operator	Meaning
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
==	Equal to
!=	Not equal to

A simple relational expression contains only one relational operator and takes the following form:

$$ae1\ relational\ operator\ ae2$$

Here ae1 and ae2 are arithmetic expressions, which may be simple constants, variables or combination of them. The value of a relational expression is either *one* or *zero.* It is one if the specified relation is true and zero if the relation is false.

E.g. 10 < 20 is true, 20 < 10 is false.

These expressions are used in decision statements such as if and while to decide the course of action of a running program.

Logical Operators: C has the following three logical operators:

Operator	Meaning
&&	Logical AND
II	Logical OR
!	Logical NOT

The logical operator && and I I are used when we want to test more than one condition & make decisions.

E.g. a>b && x == 10

An expression of this kind will combine two or more relational expressions is termed as logical expression or compound relational expression. The logical expression given below is true if a>b is true and x == 10 is true. If either or both of them are false, the expression is false.

TRUTH TABLE			
op-1	**op-2**	**op-1 && op-2**	**op-1 || op-2**
1	1	1	1
1	0	0	1
0	1	0	1
0	0	0	0

Logical NOT is used to reverse the truth value of its operand. (i.e. NOT F -> T)
An expression containing a logical operator is termed as a logical expression. A logical expression also yields a value of one or zero.

Assignment Operator: (=)
These are used to assign the result of an expression to a variable. C has a set of shorthand assignment operators of the form:

$$v\ op = exp;$$

where v is a variable, exp is an expression and op is a C arithmetic operator. The operator op= is known as shorthand assignment operator.
The above expression can be equivalent to $v = v\ op\ (exp)$;

E.g. x += y+1; -> x = x + (y+1);

Statement with simple assignment operator	Statement with shorthand operator
`a = a+1`	`a += 1`
`a = a-1`	`a -= 1`
`a = a*(n+1)`	`a *= (n+1)`
`a = a/(n+1)`	`a /= (n+1)`
`a = a%b`	`a %= b`

The use of shorthand assignment operators has 3 advantages:

1. What appears on the left-hand side need not be repeated and therefore it becomes easier to write.
2. The statement is more concise and easier to read.
3. The statement is more efficient.

Increment & Decrement Operators:
C has two very useful operators not generally found in other languages. These are increment and decrement operators: ++ and --.
The operator ++ adds 1 to the operand, while -- subtracts 1.

Pre/Post Increment/Decrement Operators
PRE means do the operation first followed by any assignment operation. POST means do the operation after any assignment operation.
++m; or m++; I --m; or m--;
++m; is equivalent to m=m+1; (or m+=1;) / --m; is equivalent to m=m-1; (or m-=1;)

While m++ and ++m mean the same when they form statements independently, they behave differently when they are used in expressions on the right-hand side of an assignment statement.
E.g. (i) m = 5;
y = ++m; This statement results y and m = 6

Since the prefix operator first adds 1 to the operand and then the result is assigned to the variable on left.

(ii) m = 5;

y = m++; This statement results y = 5 and m = 6

Since the postfix operator first assigns the value to the variable on left then increments the operand.

We use increment and decrement statements in for and while loops extensively.

Conditional Operator:

A ternary operator pair "? :" is available in C to construct conditional expressions of the form:

$$exp1 \ ? \ exp2 : exp3;$$

where exp1, exp2 and exp3 are expressions.

There operator ?: works as follows: exp1 is evaluated first. If it is true, then exp2 is evaluated and becomes the value of the expression. If exp1 is false, exp3 is evaluated and its value becomes the value of the expression.

E.g. a=10;

b=15;

x=(a>b) ? a : b; In this, the x will be assigned with the value of b.

Bitwise Operators:

In C, operations on bits at individual levels can be carried out using Bitwise operators. These are used for manipulation of data at bit level. These operators are used for testing the bits, or shifting them right or left. These may not be applied to float or double.

Operator	Meaning
&	bitwise AND
I	bitwise OR
^	bitwise exclusive OR
<<	shift left
>>	shift right
~	One's complement

The **bitwise AND** does the logical AND of the bits in each position of a number in its binary form.

0 0 1 1 1 1 0 0 &

0 0 0 0 1 1 0 1

---

0 0 0 0 1 1 0 0 : Result

The **bitwise OR** does the logical OR of the bits in each position of a number in its binary form.

0 0 1 1 1 1 0 0 I
0 0 0 0 1 1 0 1
---
0 0 1 1 1 1 0 1 : Result

The **bitwise exclusive OR** performs a logical EX-OR function or in simple term adds the two bits discarding the carry. Thus result is zero only when we have 2 zeroes or 2 ones to perform on.

0 0 1 1 1 1 0 0 ^
0 0 0 0 1 1 0 1
---
0 0 1 1 1 0 0 0 1 : Result

The **one's complement (~)** or the bitwise complement gets us the complement of a given number. Thus we get the bits inverted, for every bit 1 the result is bit 0 and conversely for every bit 0 we have a bit 1.

Bit	One's Complement
0	1
1	0

~ 0 0 1 1 1 1 0 0 --> 1 1 0 0 0 0 1 1

Two **shift operators** shift the bits in an integer variable by a specified number of positions. The << operator shifts bits to the left, and the >> operator shifts bits to the right.
The syntax for these binary operators is `x << n` and `x >> n`.

Each operator shifts the bits in x by n positions in the specified direction.

- For a **right shift**, zeros are placed in the n high-order bits of the variable;
- For a **left shift**, zeros are placed in the n low-order bits of the variable.

Here are a few examples:

Binary 00001100 (decimal 12) right-shifted by 2 evaluates to binary 00000011 (decimal 3).

Binary 00001100 (decimal 12) left-shifted by 3 evaluates to binary 01100000 (decimal 96).

Binary 00001100 (decimal 12) right-shifted by 3 evaluates to binary 00000001 (decimal 1).

Binary 00110000 (decimal 48) left-shifted by 3 evaluates to binary 10000000 (decimal 128).

Special Operators:

C supports some operators of interest such as comma operator, sizeof operator, pointer operators (& and *) and member selection operators (. and ->).

The **comma operator** can be used to link the related expressions together. A comma-linked: list of expressions are evaluated left to right and the value of right-most exp is the value of combined expression.

E.g. value = (x=10,y=5,x+y);

First 10 is assigned to x then 5 is assigned to y & finally x + y i .e. which 15 is assigned to value .

Since comma operator has lowest precedence of all operators, the parentheses are necessary.

In for loops: for(n=1,m=10;n<=m;n++,m++)

In while loops: while(c=getchar(), c!='10')

Exchanging values: t=x, x=y, y=t;

The **sizeof** is a compile time operator and when used with an operand, it returns the number of bytes the operand occupies. The operand may be a variable, a constant or a data type qualifier.

E.g. m = sizeof(sum);
n = sizeof(long int);
k = sizeof(235L);

This operator is normally used to determine the lengths of arrays and structures when their sizes are not known to the programmer. It is also used to allocate memory space dynamically to variables during execution of a program.

Precedence of Arithmetic operators

An arithmetic expression without parentheses will be evaluated from left to right using the rules of precedence of operators. There are two distinct priority levels of arithmetic operators in C:

High Priority	*, /, %
Low Priority	+, -

The basic evaluation procedure includes two left-to-right passes through the expression. During the first pass, the high priority operators (if any) are applied as they are encountered. During the second pass, the low priority operators (if any) are applied as they are encountered.

E.g. Consider a=9, b=12 and c=3

x = a - b/3+c*2-1;

1> x = 9 - 12/3 + 3*2-1
2>x = 9 - 4 + 3*2-1
3>x = 9 - 4 + 6-1
4>x = 5 + 6 - 1
5>x = 11 - 1 6> x = 10

y = a - b/(3+c)*(2-1);

1> y = 9 - 12/6 * (2-1)

2> y = 9 - 12/6 * 1

3> y = 9 - 2 * 1

4> y = 9 - 2 5> y = 7

z = a - (b/(3+c)*2)-1;

1> z = 9 - (12/(3+3) * 2) - 1

2> z = 9 - (12/6 * 2) - 1

3> z = 9 - (2 * 2) - 1

4> z = 9 - 4 - 1

5> z = 5-1 6> z = 4

Operator Precedence and Associativity

Each operator in C has a precedence associated with it. This precedence is used to determine how an expression involving more than one operator is evaluated. There are distinct levels of precedence and an operator may belong to one of the levels. The operators at the highest level of precedence are evaluated first. The operators of the same precedence are evaluated either from left to right or from right to left, depending on the level. This is known as associativity property of an operator.

`a++ a--`	left to right
`!a ~a (type)a ++a --a`	right to left!
`a*b a/b a%b`	left to right
`a+b a-b`	left to right
`a>>b a<<b`	left to right
`a>b a>=b a<b a<=b`	left to right
`a==b a!=b`	left to right
`a&b`	left to right
`a^b`	left to right
`a|b`	left to right
`&&`	left to right
`||`	left to right
`a?b:c`	right to left
`=, +=, -=, *=, /=` `%=, <<=, >>=, &=` `|=, ^=`	right to left
`,`	left to right

ARITHMETIC EXPRESSIONS

An arithmetic expression is a combination of variables, constants and operators arranged as per the syntax of the language. C can handle any complex mathematical expressions.

Arithmetic Instruction

A C arithmetic instruction consists of a variable name on the left hand side of = and variable names and constants on the right hand side of =. The variables and constants appearing on the right hand side of = are connected by arithmetic operators like +, -, *, /, and %.

$$variable - name = expression\ [or]\ value;$$

E.g.
```
int a;
a=3200;
float kot, deta, alpha=9.2, beta=3.1256, gamma=100.0;
kot = 0.0056;
deta = alpha*beta/gamma + 3.2 * 2/5;
```

Here 2,5 and 3200 are integer constants and 3.2 and 0.0056 are real constants
kot, deta, alpha, beta and gamma are real variables.

The variables and constants together are called operands that are operated upon by the arithmetic operators and the result is assigned using the assignment operator, to the variable on the left-hand side.

C arithmetic instructions are of 3 types:

(1) ***Integer mode:*** This is an arithmetic instruction in which all operands are either integer variables or integer constants.

E.g.
```
int i, king, issac, noteit;
i = i+1;
king = issac*234 + noteit-7689;
```

(2) ***Real mode:*** This is an arithmetic instruction in which all operands are either real constants or real variables.

E.g.
```
float q, a, si, princ, anoy, roi;
q = a + 23.123/4.5*0.344;
si = princ*anoy*roi/100.0;
```

(3) ***Mixed mode:*** This is an arithmetic instruction in which some of the operands are integers and some of the operands are real.

E.g.
```
float si, princ, anoy, roi, avg;
int a, b, c, num;
q = a + 23.123/4.5*0.344;
si = princ*anoy*roi/100.0;
avg = (a+b+c+num)/4;
```

The execution of an arithmetic instruction: Firstly, the right hand side is evaluated using constants and the numerical values stored in the variable names. This value is then assigned to the variable on the left-hand side.

Guidelines for Arithmetic Instructions

a) C allows only one variable on left hand side of =.
   i.e., x = k + l; is legal whereas k + l = x; is illegal.
b) An arithmetic instruction is often used for storing character constants in character variables.
   E.g. char a,b,d;
   a = 'F';
   b = 'G';
   d = '+';
   When we do this the ASCII values of the characters are stored in the variables. ASCII values are used to represent any character in memory.
c) Arithmetic operations can be performed on ints, floats and chars.
   char x,y;
   int z; x='a';
   y='b';
   z=x+y;
d) No operator is assumed to be present. It must be written explicitly.
   E.g. a = c.d.b(xy) usual arithmetic statement
   b = c*d*b*(x*y); c statement
e) Unlike other high level languages, there is no operator for performing exponentiation operation.
   E.g. a = 3**2;
   b = 3 ^ 2; statements are valid.

Evaluation of Expressions

Expressions are evaluated using an assignment statement of the form

$$variable = expression;$$

Algebraic Expression	C expression
a x b - c	a * b - c
(m+n)(x+y)	(m+n)*(x+y)
$\frac{ab}{c}$	a*b/c
$3x^2 + 2x + 1$	3*x*x + 2*x + 1
$\frac{x}{y} + c$	x/y + c

Variable is any valid C variable name. When the statement is encountered, the expression is evaluated first and the result then replaces the previous value of the variable on the left-hand side. All variables used in the expression must be assigned values before evaluation is attempted.

E.g. x = a*b-c;
y = b/c*a;
z = a-b/c + d;

The blank space around an operator is optional and adds only to improve readability. When these statements are used in a program, the variables a, b, c and d must be defined before they are used in the expressions.

Rules for evaluation of expression

1. Parenthesized sub expression from left to right is evaluated.
2. If parentheses are nested, the evaluation begins with the innermost sub-expression.
3. The precedence rule is applied in determining the order of application of operators in evaluating sub-expressions.
4. The associativity rule is applied when 2 or more operators of the same precedence level appear in a sub-expression.
5. Arithmetic expressions are evaluated from left to right using the rules of precedence.
6. When parentheses are used, the expressions within parentheses assume highest priority.

Type Casting

Typecasting concept in C language is used to modify a variable from one date type to another data type. New data type should be mentioned before the variable name or value in brackets which to be typecast.

C type casting example program:

- In the below C program, 7/5 alone will produce integer value as 1.
- So, type cast is done before division to retain float value (1.4).

```
#include <stdio.h>
int main ()
{
    float x;
    x = (float) 7/5;
    printf("%f",x);
}
```

Output: 1.400000

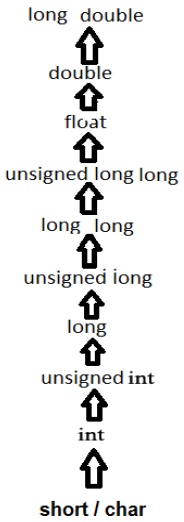
Note:

- It is best practice to convert lower data type to higher data type to avoid data loss.

- Data will be truncated when higher data type is converted to lower. For example, if float is converted to int, data which is present after decimal point will be lost.

Usual Arithmetic Conversion

The usual arithmetic conversions are implicitly performed to cast their values in a common type, C uses the rule that, in all expressions except assignments, any implicit type conversions made from a lower size type to a higher size type as shown below:



"math.h" functions

Mathematics is relatively straightforward library to use again. You must `#include<math.h>` and must remember to link in the math library at compilation:

```
cc mathprog.c -o mathprog -lm
```

Math Functions

S.no	Function	Description	Example
1	**floor()**	This function returns the nearest integer which is less than or equal to the argument passed to this function.	floor of 5.100000 is 5.000000 floor of 5.900000 is 5.000000 floor of -5.400000 is -6.000000 floor of -6.900000 is -7.000000
2	**round()**	This function returns the nearest integer value of the float/double/long double argument passed to this function. If decimal value is from ".1 to .5", it returns integer value less than the argument. If decimal value is from ".6 to .9", it returns the integer value greater than the argument.	round of 5.400000 is 5.000000 round of 5.600000 is 6.000000
3	**ceil()**	This function returns nearest integer value which is greater than or equal to the argument passed to this function.	ceil of 5.400000 is 6.000000 ceil of 5.600000 is 6.000000 Ceil of 8.33=9

4	**sin()**	This function is used to calculate sine value.	The value of sin(0.314000) : 0.308866 The value of cos(0.314000) : 0.951106 The value of tan(0.314000) : 0.324744 The value of sinh(0.250000) : 0.252612 The value of cosh(0.250000) : 1.031413 The value of tanh(0.250000) : 0.244919 The value of log(6.250000) : 1.832582 The value of log10(6.250000) : 0.795880 The value of exp(6.250000) : 518.012817
5	**cos()**	This function is used to calculate cosine.	
6	**cosh()**	This function is used to calculate hyperbolic cosine.	
7	**exp()**	This function is used to calculate the exponential "e" to the x^{th} power.	
8	**tan()**	This function is used to calculate tangent.	
9	**tanh()**	This function is used to calculate hyperbolic tangent.	
10	**sinh()**	This function is used to calculate hyperbolic sine.	
11	**log()**	This function is used to calculates natural logarithm.	
12	**log10()**	This function is used to calculates base 10 logarithm.	
13	**sqrt()**	This function is used to find square root of the argument passed to this function.	sqrt of 16 = 4.000000 sqrt of 2 = 1.414214
14	**pow()**	This is used to find the power of the given number.	2 power 4 = 16.000000 5 power 3 = 125.000000
15	**trunc()**	This function truncates the decimal value from floating point value and returns integer value.	truncated value of 16.99 = 16.000000 truncated value of 20.1 = 20.000000

Math Constants

The math.h library defines many (often neglected) constants. It is always advisable to use these definitions:

- HUGE - The maximum value of a single-precision floating-point number.
- M_E - The base of natural logarithms (e).
- M_LOG2E - The base-2 logarithm of e.
- M_LOG10E - The base-10 logarithm of e.
- M_LN2 - The natural logarithm of 2.
- M_LN10 - The natural logarithm of 10.
- M_PI - π.
- M_PI_2 - $\pi/2$.
- M_PI_4 - $\pi/4$.
- M_1_PI - $1/\pi$.
- M_2_PI - $2/\pi$.
- M_2_SQRTPI - $2/\sqrt{\pi}$.
- M_SQRT2 - The positive square root of 2.
- M_SQRT1_2 - The positive square root of 1/2.
- MAXFLOAT - The maximum value of a non-infinite single- precision floating point number.
- HUGE_VAL - positive infinity.