# Chapter 3

## Introduction to ‘C’ Programming

C is a general-purpose language which has been closely associated with the UNIX operating system for which it was developed - since the system and most of the programs that run it are written in C.

Many of the important ideas of C stem from the language BCPL, developed by Martin Richards. The influence of BCPL on C proceeded indirectly through the language B, which was written by Ken Thompson in 1970 at Bell Labs, for the first UNIX system on a DEC PDP-7. BCPL and B are "type less" languages whereas C provides a variety of data types.

In 1972 Dennis Ritchie at Bell Labs writes C and in 1978 the publication of The C Programming Language by Kernighan & Ritchie caused a revolution in the computing world.

In 1983, the American National Standards Institute (ANSI) established a committee to provide a modern, comprehensive definition of C. The resulting definition, the ANSI standard, or "ANSI C", was completed late 1988.

**Why C?**

C has been used successfully for every type of programming problem imaginable from operating systems to spreadsheets to expert systems - and efficient compilers are available for machines ranging in power from the Apple Macintosh to the Cray supercomputers. The largest measure of C's success seems to be based on purely practical considerations:

1. the portability of the compiler;
2. the standard library concept;
3. a powerful and varied repertoire of operators;
4. an elegant syntax;
5. ready access to the hardware when needed;
6. and the ease with which applications can be optimized by hand-coding isolated procedures

C is often called a "Middle Level" programming language. This is not a reflection on its lack of programming power but more a reflection on its capability to access the system's low level functions. Most high-level languages (e.g. FORTAN) provides everything the programmer might want to do already built into the language. A low level language (e.g. assembler) provides nothing other than access to the machines basic instruction set. A middle level language, such as C, probably doesn't supply all the constructs found in high-languages - but it provides you with all the building blocks that you will need to produce the results you want!

**Character Set:**

C uses the **uppercase letter** A to Z, the **lowercase letters** a to z, the **digits** 0 to 9, and certain **special characters** as building blocks to form basic program elements.(e.g. constants, variables, operators, expressions etc.). The **special characters** are listed below:

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| + | - | * | / | = | % | & | # | ! | ? | ^ | |
| " | ' | ~ | \ | \| | < | > | ( | ) | [ | ] | |
| { | } | : | ; | . | , | _ | (blank space) | | | | |

Certain **other characters** such as **@** and **$** can be **included** within **string and comments**. C uses certain combination of these characters such as \b, \n, \t to represent special conditions such as backspace, new line, and horizontal tab respectively. These character combinations are known as escape sequences.

**Identifiers**

Identifier is the word used for name given to variables, functions, constants etc. It can also be defined as a set of combinations of one or more letters or digits used to identify constants, variables, functions etc.

*Rules for Valid Identifiers*

1. ANSI Standard - 31 characters are allowed. (Some implementation of C only recognizes 8 characters). It would be better to follow the rule of 8 characters.
2. Both uppercase and lowercase letters may be used and are recognized as distinct.
3. First character should always be letter.
4. Only underscore (_) is allowed among special symbols and is treated as letter.
5. Space and other symbols are not valid.
6. Proper naming conventions should be followed.

| | |
|---|---|
| **Valid** | Count<br>test23<br>high_balance<br>_test |
| **Invalid** | $4^{th}$<br>order-no<br>error flag<br>hi!there |

**Keywords**

Keywords are the words whose meaning has already been explained to the C compiler. The keywords cannot be used as variable names because if we do so we are trying to assign a new meaning to the keyword, which is not allowed by the compiler. The keywords are also called reserve words. Note that keywords are always lowercase. There are only 32 keywords available in C

| | | | | | |
|---|---|---|---|---|---|
| auto | double | int | struct | break | else |
| long | switch | case | enum | register | typedef |
| char | extern | return | union | const | float |
| for | goto | if | short | unsigned | continue |
| signed | void | default | sizeof | do | static |
| while | volatile | | | | |

**Constants and Variables:**

**Constants**

The term constant means that it does not change during the execution of a program. It can be classified as:

a) Primary constants (integer, real, character, string)
b) Secondary constants (array, pointer, structure, union)

Here, we will discuss only primary constants.

**i. Integer constants:** Integer constants are whole numbers without any fractional parts.

*Rules for constructing Integer constants:*

1. An integer constant must have at least one digit.
2. It must not have a decimal point.
3. It could be either positive or negative

4. If no sign precedes an integer constant is assumed to be positive.
5. No commas or blank spaces are allowed within an integer constant.
6. The allowable range is -2147483648 to +2147483647 for 32 bit machine. For 16-bit machine it is -32768 to 32767.

| | |
|---|---|
| **Valid** | 1234<br>3456<br>-9746 |
| **Invalid** | 11.<br>45,35<br>$12<br>025<br>x248 |

There are three types of integers that are allowed in C

a) Decimal constants (base 10)
b) Octal constants (base 8)
c) Hexadecimal constants (base 16)

**Decimal integer constants:**

- The allowable digits in a decimal integer constant are 0,1,2,3,4,5,6,7,8,9.
- The first digit should not be 0.
- Should at least one digit.

| | |
|---|---|
| **Valid** | 1334<br>-9746 |
| **Invalid** | 11.<br>2,56<br>$78<br>025<br>x248 |

**Octal integer constants:**

- The allowable digits in a octal constant are 0,1,2,3,4,5,6,7
- Must have at least one digit and start with digit 0.

| | |
|---|---|
| **Valid** | 0245<br>-0476<br>+04013 |
| **Invalid** | 25 (does not start with 0)<br>0387( 8 is not an octal digit)<br>04.32(Decimal point is not allowed) |

**Hexadecimal integer constant:**

- The allowable digits in a hexadecimal constant are 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
- Must have at least one digit and start with 0x or 0X.

| | |
|---|---|
| **Valid** | 0x14AF<br>0X34680<br>-0x2673E |
| **Invalid** | 0345 (Must start with 0x)<br>0x45H3 (H is not a hexadecimal digit) |

**ii. Floating point / Real Constants:** A floating point constant may be written in one or two forms called fractional form or the exponent form.

*Rules for constructing floating point / real constants:*

1. A floating-point constant must have at least one digit to the left and right of the decimal point.
2. It must have a decimal point.
3. It could be either positive or negative
4. If no sign precedes an floating point constant is assumed to be positive.
5. No commas or blank spaces are allowed.
6. When value is too large or small, can be expressed in exponential form comprising mantissa and exponent. E or e should separate mantissa and exponent. The mantissa may have upto 7 digits and the exponent may be between -38 and +38.

| | |
|---|---|
| **Valid** | +325.34<br>426.0<br>32E-89<br>35e+56 |
| **Invalid** | 1 (decimal point missing)<br>1. (No digit following decimal digit)<br>-1/2 (Symbol / is illegal)<br>.5 (No digit to the left of the decimal)<br>56,8.94 (Comma not allowed)<br>-125.9e5.5 (exponent cannot be fraction)<br>0.158e+954 (exponent too large) |

**iii. character constants:** A character constant is a single alphabet, single digit, or a single special symbol enclosed within single quotes. Remember golden rule: single character single quote.

e.g. 'A' '5' '+' '?'

**iv. string constants:** A string constant is a sequence of alphanumeric characters enclosed in double quotation marks whose maximum length is 255 characters. For e.g. const char txt [ ] =”This is C and C++ Programming language.”;

e.g. "green", "My name &&&!!! ???", "$2525"

**Declaration of Constants**

- Value of an identifier defined as constant, cannot be altered during program execution.
- Generally, the constant is associated with an identifier and that name is used instead of actual number.

*General Syntax*

**const typename identifier = const;**

e.g. **const float pi=3.14159;**

**Another way to define constants** is with the #define preprocessor which has the advantage that it does not use ant storage (but who count bytes these days?).

**#define Identifier Value**

**#define PI 3.1415**

## Variables

- A variable is a named location in memory that is used to hold a value that may be modified by the program.
- A variable is an identifier that is used to represent some specified type of information within the designated portion of a program. The information (data items) must be assigned to the variable at some point in the program. The data item can then be accessed later in the program simply by referencing to the variable name.
- Thus, the information represented by the variable can change during the execution of the program. So, quantity that may vary during program execution is called variable. Each variable has a specific storage location in memory where its numerical value is stored. These locations can contain integer, real, or character constants.
- User defined variables must be declared before using them in the program.

*General syntax* typename var1, var2;

e.g. int a, b;
float c,d;
char e;

*Rules for constructing variable names*

1. A variable name is any combination of 1 to 8 alphabets, digits, or underscore. Some compilers allow variable names whose length could be up to 40 characters, still, it would be better to use the rule of 8 characters.
2. Both uppercase and lowercase letters may be used and are recognized as distinct.
3. First character should always be letter.
4. Only underscore ( _ ) is allowed among special symbols and is treated as letter.
5. Space and other symbols are not valid.

## Scope of Variables (Where Variables are declared?)

Variables will be declared in three basic places: *inside functions*, *in the definition of function parameters*, and *outside of all functions*. These are local variables, global variables, and formal parameters.

**Local Variables**

Variables that are declared inside a function are called local variables. Local variables may be referenced only by statements that are inside the block in which the variables are declared. Remember, a block of code begins with an opening curly brace and terminates with a closing curly brace. For example consider the following two functions:

```
void func1( ){
  int x;
  x=10;
}
void func2( ){
  int x;
  x=-199;
}
```

The integer variable x is declared twice, once in func1( ) and once in func2( ). The x in func1( ) has no relationship to the x in func2( ). This is because each x is only known to the code within the same block as the variable declaration.

**Global Variables**

Unlike local variables, global variables are known throughout the program and may be used by any piece of code. Also, they will hold their value throughout the program's execution. You create global variables by declaring them outside of any function.

```
#include<stdio.h>
#include<conio.h>
int count; /* count is global */
void func1( );
void func2( );
void main( ){
   count=100;
   func1( );
   getch( );
}
void func1( ){
   int temp;
   temp=count;
   func2( );
   printf("count is %d",count);
}
void func2( ){
  int count=50;
  printf("Value of count:%d",count);
}
```

Look closely at this program. Notice that although neither main() or func1() has declared the variable count, both may use it. func2(), however, has declared a local variable called count. When func2() refers to count, it refers to only its local variable, not the global one. If a global variable and a local variable have the same name, all references to that variable name inside the code block in which the local variable is declared will refer to that local variable and have no effect on the global variable. This can be convenient, but forgetting it can cause your program to act strangely, even though it looks correct.

**Formal Parameters**

If a function is to use arguments, it must declare variables that will accept the values of the arguments. These variables are called the formal parameters of the function. They behave like any other local variables inside the function.

```
#include<stdio.h>
#include<conio.h>
int square(int x);
main(){
   int t;
   t=square(t);
   printf("Value of t:%d",t);
   getch();
}
int square(int x){
   x=x*x;
   return x;
}
```

## Simple Data Types in C

C has a concept of 'data types' that are used to define a variable before its use. The definition of a variable will assign storage for the variable and define the type of data that will be held in the memory location. Broadly, Primary data types in C can be categorized as

- int
- char
- float
- double

**int:** Integers are whole numbers, both positive and negative. The keyword used to define integer is int. Example of declaring integer variable is

```
{
  int sum; sum=10;
}
```

**float:** float is used to define floating point numbers. Example of declaring float variable is

```
{
  float miles; miles=5.6;
}
```

**double:** double is used to define BIG floating point numbers. It reserves twice the storage for the numbers. Example of declaring double variable is

```
{
  double  big; big = 312E7;
}
```

**char:** char is used to define single character. Example of declaring character variable is

```
{
  char letter; letter = 'x';
}
```

*Note:* Single character use single quote.

**//Sample program illustrating each data type**

```
#include<stdio.h>
#include<conio.h>
main(){
  int sum;
  float money;
  char letter;
  double pi;
  sum = 10;
  money = 2.21;
  letter = 'A';
  pi = 2.01E6;
  printf("Value of sum=%d\n",sum);
  printf("Value of money=%f\n",money);
  printf("Value of letter=%c\n",letter);
  printf("Value of sum=%e\n",pi);
 getch( );
}
```

*Memory requirement of data type may vary from one compiler to another.*

## Modifying the Basic Data Types:

The basic types may have various modifiers preceding them. The modifiers define the amount of storage allocated to the variable. You use a modifier to alter the meaning of the base type to fit various situations more precisely. The lists of modifiers are:

- signed
- unsigned
- short
- long

signed, unsigned, short, long can be used with int.
signed, unsigned can be used with char.
long can be used with double.

- Use of signed on integers is allowed, but redundant, because the default integer declaration assumes a signed number.
- Char is unsigned by default, so signed is used to modify character.

The amount of storage allocated is not fixed. ANSI has the following rules

short int <= int <= long int
float <= double <= long double

What this means is that a "short int" should assign less than or same amount of storage as an "int" and the "int" should be less or the same bytes than a "long int".

| Type | Storage Size | Value Range | Format |
|---|---|---|---|
| **char** | 1 byte | -128 to 127 or 0 to 255 | %c |
| **unsigned char** | 1 byte | 0 to 255 | %c |
| **signed char** | 1 byte | -128 to 127 | %c |
| **int** | 2 or 4 bytes | **-32,768 to 32,767** or -2,147,483,648 to 2,147,483,647 | %d |
| **unsigned int** | 2 or 4 bytes | **0 to 65,535** or 0 to 4,294,967,295 | %lu |
| **short** | 2 bytes | -32,768 to 32,767 | %d |
| **unsigned short** | 2 bytes | 0 to 65,535 | %d |
| **long** | 4 bytes | -2,147,483,648 to 2,147,483,647 | %ld |
| **unsigned long** | 4 bytes | 0 to 4,294,967,295 | %lu |

These figures only apply to today's generation of PCs. Mainframes and midrange machines could use different figures. You can find out how much storage is allocated to a data type by using the ***sizeof* operator**.

e.g

```
main( ){
  int a;
  printf("Size of Integer is %d", sizeof a);
  getch( );
}
```

**Escape Sequences**

We saw earlier how the new line character,\n, when inserted in printf( )'s format string, takes the cursor to the beginning of the next line. The new line character is an "escape sequence", so called because the backslash symbol (\) is considered an "escape character": it causes an escape from the normal interpretation of string, so that the character is recognized as one having a special meaning.

*Some escape sequences used in C are:*

| Escape Sequence | Purpose |
|---|---|
| \n | New Line (Line feed) |
| \b | Backspace(Moves the cursor one position to the left of its current position) |
| \t | Horizontal Tab |
| \r | Carriage return( Takes the cursor to the beginning of the line in which it is currently placed) |
| \f | Form feed |
| \' | Single quote |
| \" | Double quote |
| \\ | Backslash |
| \a | Alert (Alerts the user by sounding the speaker inside the computer) |
| \? | Question mark |

e.g. printf("He said,\"Let's do it!\""); will print

```
He said,"Let's do it!"
```

# Operators

Operators are those symbols that perform some specific task. Like, when we add two numbers we generally use '+' sign. In programming also, to perform some specific task, compiler has assigned some specific symbols. These symbols are called operators. The data items that operators act upon are called operands. Some operators require one operand to act upon, whereas some require more than one. Thus, depending upon the number of operands required by the operators, we can broadly divide operators into three distinct categories.

1. Unary operators (++, -- ,-,sizeof)
2. Binary operators (Arithmetic, Assignment, Relational, Logical)
3. Tertiary operators (?:)

**1. Unary operators**

C includes a class of operators that act upon a single operand to produce a new value. Such operators are known as unary operators. Unary operators usually precede their single operand, though some unary operators are written after their operand. The most common unary operator is unary minus, where a numerical constant, variable, or expression is preceded by a minus sign.

e.g. -743, -a,-root

Hence, this operator requires single operand. The other commonly used unary operators are: ++ ( increment operator) and -- (decrement operator). The increment operator causes its operand to be incremented by 1, whereas the decrement operator causes its operand to be increased by 1.

e.g a++, a--, ++a, --a

```
main(){
    int a=5;
    printf("Value of a=%d",a);
    a++;      printf("Value of a=%d",a);
    a--;      printf("Value of a=%d",a); getch();
}
```

**Output**

```
Value of a=5
Value of a=6
Value of a=5
```

The increment and decrement operators can each be utilized in two different ways, depending upon whether the operator is written before or after the operand. If the operator precedes the operand (e.g. ++i), then the operand will be altered in value before it is utilized for its intended purpose within the program. If, however, the operator follows the operand (e.g. i++), then the value of the operand will be altered after it is utilized. Hence, increment and decrement can be pre or post.

*i++ (post increment), ++i(pre increment), i--(post decrement), --i(pre decrement)*

e.g.

```
printf("i=%d\n",i);
printf("i=%d\n",++i);
printf("i=%d\n",i);
```

**Output**

```
i=1
i=2
i=2
```

The first statement causes the original value of i to be displayed. The second statement increments i and then displays its value. The last statement displays the final value of i.

```
printf("i=%d\n",i);
printf("i=%d\n",i++);
printf("i=%d\n",i);
```

Output

```
i=1
i=1
i=2
```

The first statement causes the original value of i to be displayed. The second statement causes the current value of i to be displayed and then incremented to 2. The last statement displays the final value of i. Another unary operator is sizeof. It returns the size of operand in Bytes.

e.g.

```
main(){
    int i;
    float x;
    double d;
    char c;
  printf("integer: %d\n", sizeof i);
  printf("float: %d\n", sizeof x);
  printf("double: %d\n", sizeof d);
  printf("Character: %d\n", sizeof c);
  getch();
}
```

**Output**

```
integer: 4
float: 4
double: 8
character: 1
```

*Typecast operator* is also unary operator. It is used to change the data type of a variable.
*Syntax*: **(type) expression**
e.g. (float) x/2;

```
main(){
        int var1; float var2;
        var2= (float) var1;
}
```

Unary operators have higher precedence. Also, the associativity of the unary operator is right to left.

**2. Binary operators**

Binary operators act upon a two operands to produce a new value. Such, operators can be classified into different categories.

**Arithmetic operators -** Arithmetic operators are the operators used to perform the arithmetic operations like addition, subtraction, multiplication, division, and modulo operation. Hence five arithmetic operators are:

| | |
|---|---|
| + | for addition |
| - | for subtraction |
| * | for multiplication |
| / | for division |
| % | for remainder operation, modulo operator |

*General Structure:*
Result = operand1 (Arithmetic operator) operand2
Where, arithmetic operator could be any one of the five operators mentioned above. Once the operator performs a particular operation, result is then assigned to 'result'.

- Operands acted upon by arithmetic operators must represent numeric values. Thus, operands may be integer, float, or character quantities. Character quantities are also allowed as each character is associated with equivalent integer number ( ASCII value).
- Modulus operator requires that both operands be integer and second operator be non zero. Similarly, division operator requires the second operator be non zero.
- Arithmetic operators (except %) can be used in 3 modes:
    - Integer mode: where both the operands are integer
    - Real mode: where both the operands are real
    - Mixed mode: where one operand is integer and second operator is real
- Result is always a floating point number for real mode and mixed mode. For integer mode, result is always an integer number.
- All the operators should be explicitly written. No assumption should be made as in normal math like z=ab is not valid. We must write z=a*b

```
main(){
    int a=10,b=3; int c,d,e,f;
    c=a+b;
    d=a-b;
    e=a*b;
    f=a%b;
    printf("Value of c=%d\n",c);
    printf("Value of d=%d\n",d);
    printf("Value of e=%d\n",e);
    printf("Value of f=%d\n",f); getch();
}
```

**Assignment operators -** Assignment operators are used when the value is to be assigned to an identifier, a variable. With execution of assignment operators, value at the right is assigned to the left. The destination variable looses the old value, i.e. old value is over ridden with the new value. If previous value is also required; it should be saved in some other variables.
*General Structure* : **Destination (assignment operator) Source**

The **assignment operators** available in C are:
= for assignment
+= add with destination and assign
-= deduct from destination and assign
*= multiply with destination and assign
/= divide the destination and assign
%= assign the remainder after dividing the destination

exp1+=exp2 is equivalent to exp1=exp1+exp2
exp1-=exp2 is equivalent to exp1=exp1-exp2
exp1*=exp2 is equivalent to exp1=exp1*exp2
exp1/=exp2 is equivalent to exp1=exp1/exp2
exp1%=exp2 is equivalent to exp1=exp1%exp2

If the two operands in an assignment expression are of different data types, then the value of the expression on the right will automatically be converted to the type of identifier on the left.

```
main(){
    int i=5;
   float j=3.14;
    i=j;
    j=i;
    printf("value of i=%d\n",i);
    printf("value of j=%f\n",j); getch();
}
```

**Output**

```
value of i=3
value of j=3.0
```

Under some circumstances, this automatic type conversion can result in an alteration of the data being assigned. For example:

- A floating point value may be truncated if assigned to an integer identifier
- A double precision value may be rounded if assigned to an floating point identifier

Now suppose that j and k are both integer type variables, and that k has been assigned a value of 5. Several assignment expressions that make use of these two variables are shown below.

| Expression | Value |
|---|---|
| j=k | 5 |
| j=k/2 | 2 |
| j=2*k/2 | 5 |
| j=2*(k/2) | 4 (truncated division followed by division) |

For multiple assignment, identifier1=identifier2=identifier3, assignment operation is carried out form right to left. So, given expression is equivalent to

*Identifier1=(identifier2=identifier3)*

**Hierarchy of Operation**

Precedence and associativity comes into role when evaluating complex mathematical expression in programming. Suppose, we have an expression like:

Z=a+2*b/c*d+23+a*a;

Precedence defines which of the operator is to be executed first, like in simple rule of mathematics, BODMAS, bracket is operated first. However, evaluating expression in programming is not that simple. Though BODMAS rule is valid, we have to tale in account some other rules also.

| Priority | Operators |
|---|---|
| 1st | * / % |
| 2nd | + - |
| 3rd | = |

In case, there is a tie between operators of same priority preference is given to the one that occurs first. This is called associativity. Associativity is the preference of the operators (of same precedence level), when there is tie between them. e.g. suppose we have

**z=a+b*c**

Then, operation occurs as z=a+(b*c) as * higher precedence than +

But when the case is like

Z=a*b+c/d, operation occurs as

Z=(a*b)+(c/d) i.e. multiplication is done first. Then (c/d) is evaluated. Then only the two values are added. It happens so because * and / have higher precedence than +. However, as * and / have same precedence level, * is evaluated first, as the associativity of the arithmetic operators is from left to right, hence whichever operator occurs first, that is evaluated.

**Relational and Logical Operators**

These operators are very helpful for making decisions. Depending upon the condition, it returns either 0 or 1. When the condition with these operators is true, 1 is returned. If the condition is false, it returns 0. There are four **relational operators** in C

| | |
|---|---|
| < | less than |
| > | greater than |
| <= | less than or equals to |
| >= | greater than or equals to |

- The precedence of these operators is below the arithmetic and unary operators.
- The associativity of these operators is from left to right.

**Equality operators** are closely associated with this group

| | |
|---|---|
| = = | equal to |
| != | not equal to |

- Precedence level of equality operator is below the relational operators. These operators also have a left to right associativity.

There are two **logical operators**

| | |
|---|---|
| && | logical and |
| \|\| | logical or |

- Logical operators act upon operands that are themselves logical expressions.
- Individual logical expressions are combined into more complex conditions that are either true or false.
- For the logical operators, any non-zero positive value is considered as true.

Consider a case where int j=7; float k=5.4; char c='w';

| Expression | Interpretation | Value |
|---|---|---|
| (j>=6)&&(c= ='w') | true | 1 |
| (k<11)&&(j>100) | false | 0 |
| (j>=6)‖(c= = 19) | true | 1 |
| (c!=p)‖(j+k)<10 | true | 1 |

**Comma Operator**

- C uses comma operator(,) as the separator in declaration of the same types. e.g. int a,b,c
- It can also be used while initializing the content of an array. e.g. int a[3]={1,5,7}
- It is also used for separating expressions. e.g. for(i=0,j=1;i<j;i++,j--)

**3. Tertiary operators**

Binary operators work on two operands. Unary operators operate on single operand whereas tertiary operator requires three operands to operate. Conditional operator is a tertiary operator.

*Conditional Operator*

Simple conditional operations can be carried out with the conditional operator ( ? : ). An expression that makes use of the conditional operator is called a conditional expression. Such an expression can be written in place of the more traditional if-else statement.

A conditional expression is written in the form

*(Test expression) ? true_value: false-value;*

Example: In the conditional expression shown below, assume that j is an integer variable

*(j<0)?0:100*

The expression (j<0) is evaluated first. If it is true (i.e. if the value of j is less than 0), the entire conditional expression takes on the value 0. Otherwise, the entire conditional expression takes on the value 100.

**Operator Precedence Group**

| Operator Category | Operators | Associativity |
|---|---|---|
| Arithmetic Multiply, Divide, & Remainder | *, /, % | L->R |
| Arithmetic Add and Subtract | +, - | L->R |
| Relational Operators | <, >, <=, >= | L->R |
| Equality Operators | ==, != | L->R |
| Logical AND | && | L->R |
| Logical OR | ‖ | L->R |
| Conditional Operator | ?: | **R->L** |
| Assignment Operator | =, +=, -=, *=, /=, %= | **R->L** |
| Unary Operators | +, ++, --, sizeof (type) | **R->L** |

*Type Conversion* - Data type of one type can be converted to another type explicitly by using type cast operator. In C typecasting can be done as: **(new type) variable**;

Consider a case as:

int a; float b;

Then, b=(int) a; Besides type casting operator, there occurs default promotion or demotion of the value to be assigned during operations.

```
main( ){
   int a; float b;
   a=3.5; b=30;
}
```

Here, for a=3.5, 3.5 is demoted to an integer and hence, a is assigned an integer value of 3. Similarly, 30 is promoted to floating point value of 30.000000 and assigned to b. In this case type casting occurs automatically. Generally, source data is converted to match the data type of destination.