C programming language also allows defining various other types of variables like Enumeration, Pointer, Array, Structure, Union, etc.

Variable Definition in C

A variable definition tells the compiler where and how much storage to create for the variable. A variable definition specifies a data type and contains a list of one or more variables of that type as follows −

```
type variable_list;
```

Here, **type** must be a valid C data type including char, w_char, int, float, double, bool, or any user-defined object; and **variable_list** may consist of one or more identifier names separated by commas. Some valid declarations are shown here −

```
int    i, j, k;
char   c, ch;
float  f, salary;
double d;
```

The line **int i, j, k;** declares and defines the variables i, j, and k; which instruct the compiler to create variables named i, j and k of type int.

Variables can be initialized (assigned an initial value) in their declaration. The initializer consists of an equal sign followed by a constant expression as follows −

```
type variable_name = value;
```

Some examples are −

```
extern int d = 3, f = 5;    // declaration of d and f.
int d = 3, f = 5;           // definition and initializing d and f.
byte z = 22;                // definition and initializes z.
char x = 'x';               // the variable x has the value 'x'.
```

For definition without an initializer: variables with static storage duration are implicitly initialized with NULL (all bytes have the value 0); the initial value of all other variables are undefined.

Variable Declaration in C

A variable declaration provides assurance to the compiler that there exists a variable with the given type and name so that the compiler can proceed for further compilation without requiring the complete detail about the variable. A variable definition has its meaning at the time of compilation only; the compiler needs actual variable definition at the time of linking the program. A variable declaration is useful when multiple files are used.

OPERATORS AND EXPRESSIONS

C language offers many types of operators. They are,

1. Arithmetic operators
2. Assignment operators
3. Relational operators
4. Logical operators
5. Bit wise operators
6. Conditional operators (ternary operators)
7. Increment/decrement operators
8. Special operators

S.no	Types of Operators	Description
1	**Arithmetic operators**	These are used to perform mathematical calculations like addition, subtraction, multiplication, division and modulus
2	**Assignment operators**	These are used to assign the values for the variables in C programs.
3	**Relational operators**	These operators are used to compare the value of two variables.
4	**Logical operators**	These operators are used to perform logical

		operations on the given two variables.
5	**Bit wise operators**	These operators are used to perform bit operations on given two variables.
6	**Conditional (ternary) operators**	Conditional operators return one value if condition is true and returns another value is condition is false.
7	**Increment/decrement operators**	These operators are used to either increase or decrease the value of the variable by one.
8	**Special operators**	&, *, sizeof( ) and ternary operators.

ARITHMETIC OPERATORS IN C

C Arithmetic operators are used to perform mathematical calculations like addition, subtraction, multiplication, division and modulus in C programs.

S.no	Arithmetic Operators	Operation	Example
1	+	Addition	A+B
2	–	Subtraction	A-B
3	*	multiplication	A*B
4	/	Division	A/B
5	%	Modulus	A%B

EXAMPLE PROGRAM FOR C ARITHMETIC OPERATORS

In this example program, two values "40" and "20" are used to perform arithmetic operations such as addition, subtraction, multiplication, division, modulus and output is displayed for each operation.

```
#include <stdio.h>
 int main()
{
int a=40,b=20, add,sub,mul,div,mod;
add = a+b;
sub = a-b;
mul = a*b;
div = a/b;
mod = a%b;
printf("Addition of a, b is : %d\n", add);
printf("Subtraction of a, b is : %d\n", sub);
printf("Multiplication of a, b is : %d\n", mul);
printf("Division of a, b is : %d\n", div);
printf("Modulus of a, b is : %d\n", mod);
}
```

OUTPUT:

```
Addition of a, b is : 60
Subtraction of a, b is : 20
Multiplication of a, b is : 800
Division of a, b is : 2
Modulus of a, b is : 0
```

ASSIGNMENT OPERATORS IN C

In C programs, values for the variables are assigned using assignment operators.
For example, if the value "10" is to be assigned for the variable "sum", it can be assigned as "sum = 10;"

Other assignment operators in C language are given below.

Operators		Example	Explanation
Simple assignment operator	=	sum = 10	10 is assigned to variable sum
	+=	sum += 10	This is same as sum = sum + 10
	-=	sum -= 10	This is same as sum = sum – 10
	*=	sum *= 10	This is same as sum = sum * 10
	/+	sum /= 10	This is same as sum = sum / 10
	%=	sum %= 10	This is same as sum = sum % 10
	&=	sum&=10	This is same as sum = sum & 10
Compound assignment operators	^=	sum ^= 10	This is same as sum = sum ^ 10

EXAMPLE PROGRAM FOR C ASSIGNMENT OPERATORS:

In this program, values from 0 – 9 are summed up and total "45" is displayed as output. Assignment operators such as "=" and "+=" are used in this program to assign the values and to sum up the values.

```
# include <stdio.h>
 int main()
{
int Total=0,i;
for(i=0;i<10;i++)
{
Total+=i; // This is same as Total = Toatal+i
}
printf("Total = %d", Total);
}
```

OUTPUT:

```
Total = 45
```

RELATIONAL OPERATORS IN C

Relational operators are used to find the relation between two variables. i.e. to compare the values of two variables in a C program.

S.no	Operators	Example	Description
1	>	x > y	x is greater than y
2	<	x < y	x is less than y
3	>=	x >= y	x is greater than or equal to y
4	<=	x <= y	x is less than or equal to y

5	==	x == y	x is equal to y
6	!=	x != y	x is not equal to y

EXAMPLE PROGRAM FOR RELATIONAL OPERATORS IN C

In this program, relational operator (==) is used to compare 2 values whether they are equal are not.

If both values are equal, output is displayed as " values are equal". Else, output is displayed as "values are not equal".

Note: double equal sign (==) should be used to compare 2 values. We should not single equal sign (=).

```
#include <stdio.h>

 int main()

{

int m=40,n=20;

if (m == n)

{

printf("m and n are equal");

}

else

{

printf("m and n are not equal");

}

}
```

OUTPUT:

```
m and n are not equal
```

LOGICAL OPERATORS IN C

These operators are used to perform logical operations on the given expressions.
There are 3 logical operators in C language. They are, logical AND (&&), logical OR (||) and logical NOT (!).

S.no	Operators	Name	Example	Description
1	&&	logical AND	(x>5)&&(y<5)	It returns true when both conditions are true
2	||	logical OR	(x>=10)||(y>=10)	It returns true when at-least one of the condition is true
3	!	logical NOT	!((x>5)&&(y<5))	It reverses the state of the operand "((x>5) && (y<5))" If "((x>5) && (y<5))" is true, logical NOT operator makes it false

EXAMPLE PROGRAM FOR LOGICAL OPERATORS IN C:

```
#include <stdio.h>

 int main()

{
```

```
int m=40,n=20;

int o=20,p=30;

if (m>n && m !=0)

{

printf("&& Operator : Both conditions are true\n");

}

if (o>p || p!=20)

{

printf("|| Operator : Only one condition is true\n");

}

if (!(m>n && m !=0))

{

printf("! Operator : Both conditions are true\n");

}

else

{

printf("! Operator : Both conditions are true. " \

"But, status is inverted as false\n");

}

}
```

OUTPUT:

```
&& Operator : Both conditions are true
|| Operator : Only one condition is true
! Operator : Both conditions are true. But, status is inverted as false
```

In this program, operators (&&, || and !) are used to perform logical operations on the given expressions.

&& operator – "if clause" becomes true only when both conditions (m>n and m! =0) is true. Else, it becomes false.

|| Operator – "if clause" becomes true when any one of the condition (o>p || p!=20) is true. It becomes false when none of the condition is true.

! Operator – It is used to reverses the state of the operand.

If the conditions (m>n && m!=0) is true, true (1) is returned. This value is inverted by "!" operator.

So, "! (m>n and m! =0)" returns false (0).

BIT WISE OPERATORS IN C

These operators are used to perform bit operations. Decimal values are converted into binary values which are the sequence of bits and bit wise operators work on these bits.

Bit wise operators in C language are & (bitwise AND), | (bitwise OR), ~ (bitwise OR), ^ (XOR), << (left shift) and >> (right shift).

TRUTH TABLE FOR BIT WISE OPERATION BIT WISE OPERATORS

x	y	x|y	x & y	x ^ y
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	0

Operator_symbol	Operator_name
&	Bitwise_AND
|	Bitwise OR
~	Bitwise_NOT
^	XOR
<<	Left Shift
>>	Right Shift

Consider x=40 and y=80. Binary form of these values are given below.

x = 00101000

y= 01010000

All bit wise operations for x and y are given below.
x&y = 00000000 (binary) = 0 (decimal)
x|y = 01111000 (binary) = 120 (decimal)
~x = 11111111111111111111111111 11111111111111111111111111111111111010111
.. ..= -41 (decimal)
x^y = 01111000 (binary) = 120 (decimal)
x << 1 = 01010000 (binary) = 80 (decimal)
x >> 1 = 00010100 (binary) = 20 (decimal)

Note:
Bit wise NOT: Value of 40 in binary is0000000000000000000000000000000000000000000000000010100000000000. So, all 0's are converted into 1's in bit wise NOT operation.
Bit wise left shift and right shift : In left shift operation "x << 1 ", 1 means that the bits will be left shifted by one place. If we use it as "x << 2 ", then, it means that the bits will be left shifted by 2 places.

EXAMPLE PROGRAM FOR BIT WISE OPERATORS IN C
In this example program, bit wise operations are performed as shown above and output is displayed in decimal format.

```
#include <stdio.h>

int main()

{

int m = 40,n = 80,AND_opr,OR_opr,XOR_opr,NOT_opr ;

AND_opr = (m&n);

OR_opr = (m|n);

NOT_opr = (~m);

XOR_opr = (m^n);

printf("AND_opr value = %d\n",AND_opr );

printf("OR_opr value = %d\n",OR_opr );

printf("NOT_opr value = %d\n",NOT_opr );
printf("XOR_opr value = %d\n",XOR_opr );

printf("left_shift value = %d\n", m << 1);

printf("right_shift value = %d\n", m >> 1);

}
```

OUTPUT:

```
AND_opr value = 0
OR_opr value = 120
NOT_opr value = -41
XOR_opr value = 120
left_shift value = 80
right_shift value = 20
```

CONDITIONAL OR TERNARY OPERATORS IN C

Conditional operators return one value if condition is true and returns another value is condition is false.

This operator is also called as ternary operator.

Syntax : (Condition? true_value: false_value);

Example : (A > 100 ? 0 : 1);

In above example, if A is greater than 100, 0 is returned else 1 is returned. This is equal to if else conditional statements.

EXAMPLE PROGRAM FOR CONDITIONAL/TERNARY OPERATORS IN C

```
#include <stdio.h>

 int main()

{

int x=1, y ;

y = ( x ==1 ? 2 : 0 ) ;

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

printf("y value is %d", y);
}
```

OUTPUT:

```
x value is 1
y value is 2
```

C – Increment/decrement Operators
PREVNEXT

Increment operators are used to increase the value of the variable by one and decrement operators are used to decrease the value of the variable by one in C programs.

Syntax:

Increment operator: ++var_name ;(or) var_name++;

Decrement operator: – -var_name; (or) var_name – -;

Example:

Increment operator : ++ i ; i ++ ;

Decrement operator : – – i ; i – – ;

EXAMPLE PROGRAM FOR INCREMENT OPERATORS IN C

In this program, value of "i" is incremented one by one from 1 up to 9 using "i++" operator and output is displayed as "1 2 3 4 5 6 7 8 9".

```
//Example for increment operators

#include <stdio.h>

int main()

{

    int i=1;

    while(i<10)

    {

       printf("%d ",i);

       i++;

    }

}
```

OUTPUT:

```
1 2 3 4 5 6 7 8 9
```

EXAMPLE PROGRAM FOR DECREMENT OPERATORS IN C

In this program, value of "I" is decremented one by one from 20 up to 11 using "i–" operator and output is displayed as "20 19 18 17 16 15 14 13 12 11".

```
//Example for decrement operators

#include <stdio.h>

int main()

{

   int i=20;

   while(i>10)

   {

      printf("%d ",i);

      i--;

   }

}
```

OUTPUT:

```
20 19 18 17 16 15 14 13 12 11
```

DIFFERENCE BETWEEN PRE/POST INCREMENT & DECREMENT OPERATORS IN C

Below table will explain the difference between pre/post increment and decrement operators in C.

S.no	Operator type	Operator	Description
1	Pre increment	++i	Value of i is

			incremented before assigning it to variable i.
2	Post–increment	i++	Value of i is incremented after assigning it to variable i.
3	Pre decrement	— –i	Value of i is decremented before assigning it to variable i.
4	Post_decrement	i– —	Value of i is decremented after assigning it to variable i.

EXAMPLE PROGRAM FOR PRE – INCREMENT OPERATORS IN C

```
//Example for increment operators

#include <stdio.h>

int main()

{

        int i=0;

        while(++i < 5 )

        {

            printf("%d ",i);

        }

        return 0;

}
```

OUTPUT:

```
1 2 3 4
```

Step 1 : In above program, value of "i" is incremented from 0 to 1 using pre-increment operator.

Step 2 : This incremented value "1" is compared with 5 in while expression.

Step 3 : Then, this incremented value "1" is assigned to the variable "i".

Above 3 steps are continued until while expression becomes false and output is displayed as "1 2 3 4".

EXAMPLE PROGRAM FOR POST – INCREMENT OPERATORS IN C

```
#include <stdio.h>

int main()

{

        int i=0;

        while(i++ < 5 )

        {

            printf("%d ",i);

        }

        return 0;

}
```

OUTPUT:

```
1 2 3 4 5
```

Step 1 : In this program, value of i "0" is compared with 5 in while expression.

Step 2 : Then, value of "i" is incremented from 0 to 1 using post-increment operator.

Step 3 : Then, this incremented value "1" is assigned to the variable "i".

Above 3 steps are continued until while expression becomes false and output is displayed as "1 2 3 4 5".

EXAMPLE PROGRAM FOR PRE – DECREMENT OPERATORS IN C

```
#include <stdio.h>

int main()

{

        int i=10;

        while(--i > 5 )

        {

            printf("%d ",i);

        }

        return 0;

}
```

OUTPUT:

```
9 8 7 6
```

Step 1 : In above program, value of "i" is decremented from 10 to 9 using pre-decrement operator.
Step 2 : This decremented value "9" is compared with 5 in while expression.
Step 3 : Then, this decremented value "9" is assigned to the variable "i".
Above 3 steps are continued until while expression becomes false and output is displayed as "9 8 7 6".

EXAMPLE PROGRAM FOR POST – DECREMENT OPERATORS IN C:

```
#include <stdio.h>

int main()

{

        int i=10;

        while(i-- > 5 )

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

OUTPUT:

```
9 8 7 6 5
```

Step 1 : In this program, value of i "10" is compared with 5 in while expression.
Step 2 : Then, value of "i" is decremented from 10 to 9 using post-decrement operator.
Step 3 : Then, this decremented value "9" is assigned to the variable "i".
Above 3 steps are continued until while expression becomes false and output is displayed as "9 8 7 6 5".

SPECIAL OPERATORS IN C:

Below are some of special operators that C language offers.

S.no	Operators	Description
1	&	This is used to get the address of the variable. Example : &a will give address of a.
2	*	This is used as pointer to a variable. Example : * a where, * is pointer to the variable a.
3	Sizeof ()	This gives the size of the variable. Example : size of (char) will give us 1.

EXAMPLE PROGRAM FOR & AND * OPERATORS IN C

In this program, "&" symbol is used to get the address of the variable and "*" symbol is used to get the value of the variable that the pointer is pointing to. Please refer **C – pointer** topic to know more about pointers.

```
#include <stdio.h>

int main()

{

int *ptr, q;

q = 50;

/* address of q is assigned to ptr */

ptr = &q;

/* display q's value using ptr variable */

printf("%d", *ptr);

return 0;

}
```

OUTPUT:

```
50
```

EXAMPLE PROGRAM FOR SIZEOF() OPERATOR IN C

sizeof() operator is used to find the memory space allocated for each C data types.

```
#include <stdio.h>

#include <limits.h>

 int main()

{

int a;

char b;
float c;

double d;

printf("Storage size for int data type:%d \n",sizeof(a));

printf("Storage size for char data type:%d \n",sizeof(b));

printf("Storage size for float data type:%d \n",sizeof(c));

printf("Storage size for double data type:%d\n",sizeof(d));

return 0;

}
```

OUTPUT:

```
Storage size for int data type:4
Storage size for char data type:1
Storage size for float data type:4
Storage size for double data type:8
```

EXPRESSIONS

Arithmetic expression in C is a combination of variables, constants and operators written in a proper syntax. C can easily handle any complex mathematical expressions but these mathematical expressions have to be written in a proper syntax. Some examples of mathematical expressions written in proper syntax of C are

Note: C does not have any operator for exponentiation.

C OPERATOR PRECEDENCE AND ASSOCIATIVITY

C operators in order of *precedence* (highest to lowest). Their associativity indicates in what order operators of equal precedence in an expression are applied.

Operator	Description	Associativity
( )	Parentheses (function call) (see Note 1)	left-to-right
[ ]	Brackets (array subscript)	
.	Member selection via object name	
->	Member selection via pointer	
++ --	Postfix increment/decrement (see Note 2)	

++ -- + - ! ~ (*type*) * & sizeof	Prefix increment/decrement Unary plus/minus Logical negation/bitwise complement Cast (convert value to temporary value of *type*) Dereference Address (of operand) Determine size in bytes on this implementation	right-to-left
* / %	Multiplication/division/modulus	left-to-right
+ -	Addition/subtraction	left-to-right
<< >>	Bitwise shift left, Bitwise shift right	left-to-right
< <= > >=	Relational less than/less than or equal to Relational greater than/greater than or equal to	left-to-right
== !=	Relational is equal to/is not equal to	left-to-right
&	Bitwise AND	left-to-right
^	Bitwise exclusive OR	left-to-right
|	Bitwise inclusive OR	left-to-right
&&	Logical AND	left-to-right
||	Logical OR	left-to-right
? :	Ternary conditional	right-to-left
= += -= *= /= %= &= ^= |= <<= >>=	Assignment Addition/subtraction assignment Multiplication/division assignment Modulus/bitwise AND assignment Bitwise exclusive/inclusive OR assignment Bitwise shift left/right assignment	right-to-left
,	Comma (separate expressions)	left-to-right

Note 1:

Parentheses are also used to group sub-expressions to force a different precedence; such parenthetical expressions can be nested and are evaluated from inner to outer.

Note 2:

Postfix increment/decrement have high precedence, but the actual increment or decrement of the operand is delayed (to be accomplished sometime before the statement completes execution). So in the statement **y = x * z++;** the current value of **z** is used to evaluate the expression (*i.e.*, **z++** evaluates to **z**) and **z** only incremented after all else is done.

EVALUATION OF EXPRESSION

At first, the expressions within parenthesis are evaluated. If no parenthesis is present, then the arithmetic expression is evaluated from left to right. There are two priority levels of operators in C.

High priority: * / %

Low priority: + -

The evaluation procedure of an arithmetic expression includes two left to right passes through the entire expression. In the first pass, the high priority operators are applied as they are encountered and in the second pass, low priority operations are applied as they are encountered. Suppose, we have an arithmetic expression as:

```
x = 9 – 12 / 3 + 3 *2 - 1
```

This expression is evaluated in two left to right passes as:

First Pass

```
Step 1: x = 9-4 + 3 * 2 – 1
Step 2: x = 9 – 4 + 6 – 1
```

Second Pass

```
Step 1: x = 5 + 6 – 1
Step 2: x = 11 – 1
Step 3: x = 10
```

But when parenthesis is used in the same expression, the order of evaluation gets changed.

For example,

```
x = 9 – 12 / (3 + 3) * (2 – 1)
```

When parentheses are present then the expression inside the parenthesis are evaluated first from left to right. The expression is now evaluated in three passes as:

First Pass

```
Step 1: x = 9 – 12 / 6 * (2 – 1)
Step 2: x= 9 – 12 / 6 * 1
```

Second Pass

```
Step 1: x= 9 – 2 * 1
Step 2: x = 9 – 2
```

Third Pass

Step 3: x= 7

There may even arise a case where nested parentheses are present (i.e. parenthesis inside parenthesis). In such case, the expression inside the innermost set of parentheses is evaluated first and then the outer parentheses are evaluated.

For example, we have an expression as:

x = 9 – ((12 / 3) + 3 * 2) – 1

The expression is now evaluated as:

First Pass:

Step 1: x = 9 – (4 + 3 * 2) – 1

Step 2: x= 9 – (4 + 6) – 1

Step 3: x= 9 – 10 -1

Second Pass

Step 1: x= - 1 – 1

Step 2: x = -2

Note: The number of evaluation steps is equal to the number of operators in the arithmetic expression.

TYPE CONVERSION IN EXPRESSIONS

When variables and constants of different types are combined in an expression then they are converted to same data type. The process of converting one predefined type into another is called type conversion.

Type conversion in c can be classified into the following two types:

Implicit Type Conversion

When the type conversion is performed automatically by the compiler without programmer's intervention, such type of conversion is known as **implicit type conversion** or **type promotion**.

The compiler converts all operands into the data type of the largest operand.

The sequence of rules that are applied while evaluating expressions are given below:

All short and char are automatically converted to int, then,

If either of the operand is of type long double, then others will be converted to long double and result will be long double.

Else, if either of the operand is double, then others are converted to double.

Else, if either of the operand is float, then others are converted to float.

Else, if either of the operand is unsigned long int, then others will be converted to unsigned long int.

Else, if one of the operand is long int, and the other is unsigned int, then

if a long int can represent all values of an unsigned int, the unsigned int is converted to long int.

otherwise, both operands are converted to unsigned long int.

Else, if either operand is long int then other will be converted to long int.

Else, if either operand is unsigned int then others will be converted to unsigned int.

It should be noted that the final result of expression is converted to type of variable on left side of assignment operator before assigning value to it.

Also, conversion of float to int causes truncation of fractional part, conversion of double to float causes rounding of digits and the conversion of long int to int causes dropping of excess higher order bits.

Explicit Type Conversion

The type conversion performed by the programmer by posing the data type of the expression of specific type is known as explicit type conversion.

The explicit type conversion is also known as **type casting**.

Type casting in c is done in the following form:

(data_type)expression;

where, *data_type* is any valid c data type, and *expression* may be constant, variable or expression.

For example, x=(int)a+b*d;

The following rules have to be followed while converting the expression from one type to another to avoid the loss of information:

All integer types to be converted to float.

All float types to be converted to double.

All character types to be converted to integer.

FORMATTED INPUT AND OUTPUT

The C Programming Language is also called the Mother of languages. The C language was developed by Dennis Ritchie between 1969 and 1973 and is a second and third generation of languages. The C language provides both low and high level features it provides both the power of low-level languages and the flexibility and simplicity of high-level languages.

C provides standard functions scanf() and printf(), for performing formatted input and output. These functions accept, as parameters, a format specification string and a list of variables.

The format specification string is a character string that specifies the data type of each variable to be input or output and the size or width of the input and output.

Now to discuss formatted output in functions.

Formatted Output

The function printf() is used for formatted output to standard output based on a format specification. The format specification string, along with the data to be output, are the parameters to the printf() function.

Syntax:

printf (format, data1, data2,……..);

In this syntax format is the format specification string. This string contains, for each variable to be output, a specification beginning with the symbol % followed by a character called the conversion character.

Example:

printf ("%c", data1);

The character specified after % is called a conversion character because it allows one data type to be converted to another type and printed.

See the following table conversion character and their meanings.

Conversion Character	**Meaning**
d	The data is converted to decimal (integer)
c	The data is taken as a character.
s	The data is a string and character from the string , are printed until a NULL, character is reached.

f	The data is output as float or double with a default Precision 6.
Symbols	**Meaning**
\n	For new line (linefeed return)
\t	For tab space (equivalent of 8 spaces)

Example

```
printf ("%c\n",data1);
```

The format specification string may also have text.

Example

```
printf ("Character is:"%c\n", data1);
```

The text "Character is:" is printed out along with the value of data1.

Example with program

```
#include<stdio.h>
#include<conio.h>
Main()
{
Char alphabh="A";
int number1= 55;
float number2=22.34;
printf("char= %c\n",alphabh);
printf("int= %d\n",number1);
printf("float= %f\n",number2);
getch();
clrscr();
retrun 0;
}
```

Output Here…

```
char =A
int= 55
flaot=22.340000
```

What is the output of the statement?

printf("Integer is: %d; Alphabet is:%c\n",number1, alpha);

Where number1 contains 44 and alpha contains "Krishna Singh".

Give the answer below.

Between the character % and the conversion character, there may be:

- A minus sign: Denoting left adjustment of the data.
- A digit: Specifying the minimum width in which the data is to be output, if the data has a larger number of characters then the specified width occupied by the output is larger. If the data consists of fewer characters then the specified width, it is padded to the right or to the left (if minus sign is not specified) with blanks. If the digit is prefixed with a zero, the padding is done with zeros instead of blanks.
- A period: Separating the width from the next digit.
- A digit following the period: specifying the precision (number of decimal places for numeric data) or the maximum number of characters to be output.
- Letter 1: To indicate that the data item is a long integer and not an int.

Format specification string	**Data**	**Output**
|%2d|	9	|9|
|%2d|	123	|123|
|%03d|	9	|009|
|%-2d|	7	|7|
|%5.3d|	2	|002|
|%3.1d|	15	|15|
|%3.5d|	15	|0015|
|%5s|	"Output sting"	|Output string|
|%15s|	"Output sting"	|Output string|
|%-15s|	"Output sting"	|Output string|
|%15.5s|	"Output sting"	|Output string|
|%.5s|	"Output sting"	|Output|

|%15.5s|	"Output sting"	|Output|
|%f|	87.65	|87.650000|
|%.4.1s|	87.65	|87.71|

Example based on the conversion character:

```
#include<stdio.h>
#include<conio.h>
main()
{
Int num=65;
printf("Value of num is : %d\n:, num);
printf("Character equivalent of %d is %c\n", num , num);
getch();
clrscr();
rerum o;
}
```

Output Here…

```
char =A
int= 55
flaot=22.340000
```

Formatted Input

The function scanf() is used for formatted input from standard input and provides many of the conversion facilities of the function printf().

Syntax

scanf (format, num1, num2,……);

The function scnaf() reads and converts characters from the standards input depending on the format specification string and stores the input in memory locations represented by the other arguments (num1, num2,….).

For Example:

scanf(" %c %d",&Name, &Roll No);

Note: the data names are listed as &Name and &Roll No instead of Name and Roll No respectively. This is how data names are specified in a scnaf() function. In case of string type data names, the data name is not preceded by the character &.

Example with program

Write a function to accept and display the element number and the weight of a proton. The element number is an integer and weight is fractional.

Solve here:

```
#include<stdio.h>
#include<conio.h>
main()
{
Int e_num;
Float e_wt;
printf ("Enter the Element No. and Weight of a Proton\n");
scanf ("%d %f",&e_num, &e_wt);
printf ("The Element No.is:",e_num);
printf ("The Weight of a Proton is: %f\n", e_wt);
getch();
return 0;
}
```

CONTROL STRUCTURES, ARRAYS AND STRINGS

DECISION STATEMENTS

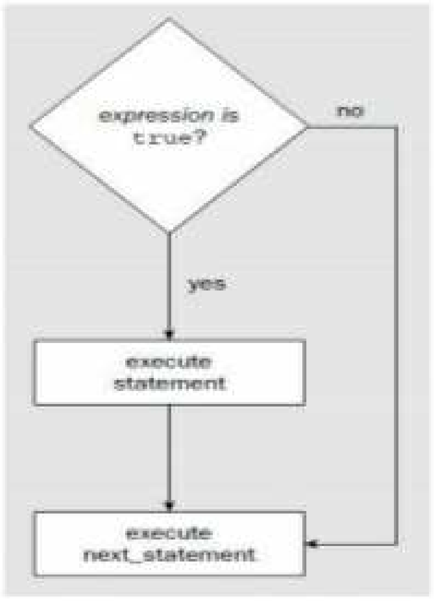
If statement:

Syntax :

```
if(expression)
  statement1;
```

Explanation :

- Expression is Boolean Expression
- It may have true or false value



Meaning of If Statement :

- It Checks whether the given Expression is Boolean or not !!
- If Expression is True Then it executes the statement otherwise jumps to next_instruction

Sample Program Code :

```
void main()
{
int a=5,b=6,c;
 c = a + b ;
 if (c==11)
     printf("Execute me 1");

 printf("Execute me 2");
}
```

Output :

```
Execute me 1
```

If Statement :

```
if(conditional)
{
   Statement No 1
   Statement No 2
   Statement No 3
   .
   .
   .
   Statement No N
}
```

Note :

More than One Conditions can be Written inside If statement.

1. Opening and Closing Braces are required only when "Code" after if statement occupies multiple lines.

```
if(conditional)
   Statement No 1
Statement No 2
Statement No 3
```

In the above example only Statement 1 is a part of if Statement.

1. Code will be executed if condition statement is True.
2. Non-Zero Number Inside if means **"TRUE Condition"**

```
if(100)
   printf("True Condition");
```

if-else Statement :

We can use if-else statement in c programming so that we can check any condition and depending on the outcome of the condition we can follow appropriate path. We have true path as well as false path.

Syntax :

```
if(expression)
  {
  statement1;
  statement2;
  }
 else
  {
  statement1;
  statement2;
  }
```

next_statement;

Explanation :

If expression is True then Statement1 and Statement2 are executed

Otherwise Statement3 and Statement4 are executed.

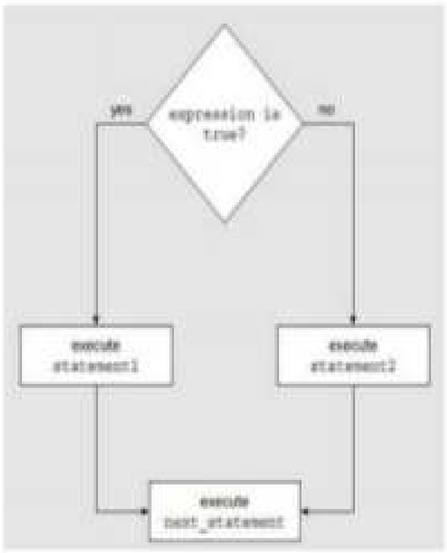
Sample Program on if-else Statement :

```
void main()
{
int marks=50;
 if(marks>=40)
  {
  printf("Student is Pass");
  }
 else
  {
  printf("Student is Fail");
  }
}
```

Output :

```
Student is Pass
```

Flowchart : If Else Statement



Consider Example 1 with Explanation:

Consider Following Example –

```
int num = 20;

if(num == 20)
   {
   printf("True Block");
   }
else
   {
   printf("False Block");
   }
```

If part Executed if Condition Statement is True.

```
if(num == 20)

   {

   printf("True Block");

   }
```

True Block will be executed if condition is True.

Else Part executed if Condition Statement is False.

```
else

   {

   printf("False Block");

   }
```

Consider Example 2 with Explanation :

More than One Conditions can be Written inside If statement.

```
int num1 = 20;
int num2 = 40;

if(num1 == 20 && num2 == 40)
  {
  printf("True Block");
  }
```

Opening and Closing Braces are required only when "Code" after if statement occupies multiple lines. Code will be executed if condition statement is True. Non-Zero Number Inside if means **"TRUE Condition"**

If-Else Statement :

```
if(conditional)
{
//True code
}
else
{
//False code
}
```

Note :

Consider Following Example –

```
int num = 20;

if(num == 20)
  {
  printf("True Block");
  }
else
  {
  printf("False Block");
  }
```

If part Executed if Condition Statement is True.

```
if(num == 20)

  {

  printf("True Block");

  }
```

True Block will be executed if condition is True.

Else Part executed if Condition Statement is False.

```
else
   {
   printf("False Block");
   }
```

More than One Conditions can be Written inside If statement.

```
int num1 = 20;
int num2 = 40;

if(num1 == 20 && num2 == 40)
   {
   printf("True Block");
   }
```

Opening and Closing Braces are required only when "Code" after if statement occupies multiple lines.

Code will be executed if condition statement is True.

Non-Zero Number Inside if means **"TRUE Condition"**

Switch statement

Why we should use Switch Case?

- One of the classic problem encountered in **nested if-else / else-if ladder**is called **problem of Confusion**.
- It occurs when no matching else is available for if .
- As the number of **alternatives increases the Complexity of program increases drastically**.
- To overcome this , C Provide a **multi-way decision statement** called '**Switch Statement**'

See how difficult is this scenario?

```
if(Condition 1)
  Statement 1
else
  {
  Statement 2
  if(condition 2)
    {
    if(condition 3)
```

```
      statement 3
     else
        if(condition 4)
           {
           statement 4
           }
     }
  else
    {
    statement 5
    }
  }
```

First Look of Switch Case

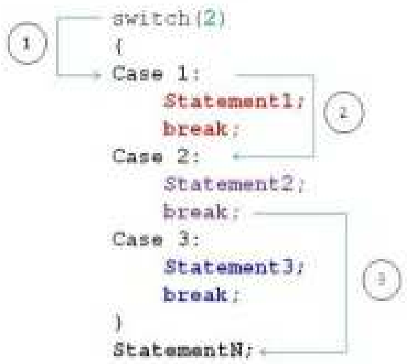
```
switch(expression)
{
case value1 :
   body1
   break;

case value2 :
   body2
   break;

case value3 :
   body3
   break;

default :
   default-body
   break;
}
next-statement;
```

Flow Diagram :



```
switch(2)
{
Case 1:
     Statement1;
     break;
Case 2:
     Statement2;
     break;
Case 3:
     Statement3;
     break;
}
StatementN;
```

*Steps are Shown in Circles.

How it works?

- Switch case checks the value of expression/variable against the list of case values and when the match is found , **the block of statement associated with that case is executed**
- Expression should be Integer **Expression / Character**
- **Break statement takes** control out of the case.
- Break Statement is **Optional**.

```
#include<stdio.h>
void main()
{
int roll = 3 ;
switch ( roll )
{
case 1:
printf ( " I am Pankaj ");
break;
case 2:
printf ( " I am Nikhil ");
break;
case 3:
printf ( " I am John ");
break;
default :
printf ( "No student found");
break;
}
}
```

As explained earlier –

3 is assigned to integer variable '**roll**'

On line 5 switch case decides – "**We have to execute block of code specified in 3rd case**".

Switch Case executes code from top to bottom.

It will now enter into first Case [i.e case 1:]

It will validate **Case number** with variable **Roll**.

If no match found then it will jump to Next Case..

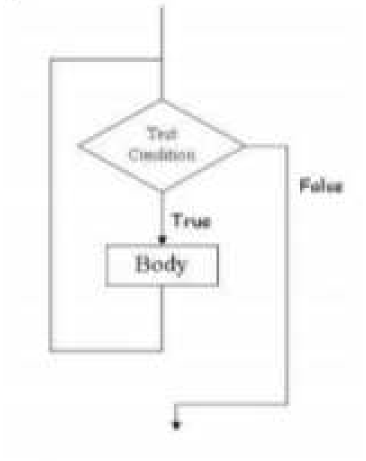
When it finds matching case it will execute block of code specified in that case.

LOOP CONTROL STATEMENTS

While statement:

While Loop Syntax:

```
initialization;
while(condition)
{
----------
----------
---------
incrementation;
}
```



Note :

For Single Line of Code – Opening and Closing braces are not needed.

while(1) is used for Infinite Loop

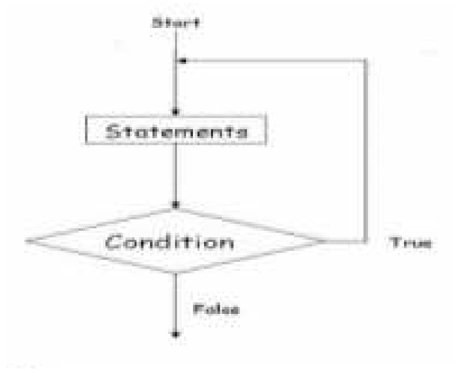
Initialization , Incrementation and Condition steps are on different Line.

While Loop is also Entry Controlled Loop.[i.e conditions are checked if found true then and then only code is executed]

Do while:

Do-While Loop Syntax :

```
initialization;
do
{
--------------
--------------
--------------
--------------
incrementation;
}while(condition);
```



Note :

It is Exit Controlled Loop.

Initialization , Incrementation and Condition steps are on different Line.

It is also called Bottom Tested [i.e Condition is tested at bottom and Body has to execute at least once]

For statement:

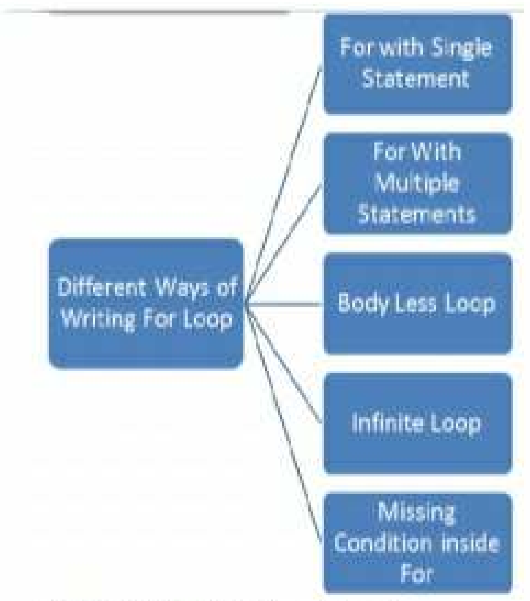
We have already seen the basics of Looping Statement in C. C Language provides us different kind of looping statements such as For loop, while loop and do-while loop. In this chapter we will be learning different flavors of for loop statement.

Different Ways of Using For Loop in C Programming

In order to do certain actions multiple times, we use loop control statements.

For loop can be implemented in different verities of using for loop –

- Single Statement inside For Loop
- Multiple Statements inside For Loop
- No Statement inside For Loop
- Semicolon at the end of For Loop
- Multiple Initialization Statement inside For
- Missing Initialization in For Loop
- Missing Increment/Decrement Statement
- Infinite For Loop
- Condition with no Conditional Operator.


Different Ways of Writing For Loop
For with Single Statement
For With Multiple Statements
Body Less Loop
Infinite Loop
Missing Condition inside For

Different Ways of Writing For Loop in C Programming Language

Way 1 : Single Statement inside For Loop

```
for(i=0;i<5;i++)
  printf("Hello");
```

Above code snippet will print Hello word 5 times.

We have single statement inside for loop body.

No need to wrap printf inside opening and closing curly block.

Curly Block is Optional.

Way 2 : Multiple Statements inside For Loop

```
for(i=0;i<5;i++)
   {
   printf("Statement 1");
   printf("Statement 2");
   printf("Statement 3");

   if(condition)
       {
       --------
       --------
       }
   }
```

If we have block of code that is to be executed multiple times then we can use curly braces to wrap multiple statement in for loop.

Way 3 : No Statement inside For Loop

```
for(i=0;i<5;i++)
  {

  }
```

this is bodyless for loop. It is used to increment value of "i".This verity of for loop is not used generally.

At the end of above for loop value of i will be 5.

Way 4 : Semicolon at the end of For Loop

```
for(i=0;i<5;i++);
```

Generally beginners thought that , we will get compile error if we write semicolon at the end of for loop.

This is perfectly legal statement in C Programming.

This statement is similar to bodyless for loop. (Way 3)

Way 5 : Multiple Initialization Statement inside For

```
for(i=0,j=0;i<5;i++)
  {
  statement1;
  statement2;
  statement3;
  }
```

Multiple initialization statements must be seperated by Comma in for loop.

Way 6 : Missing Increment/Decrement Statement

```
for(i=0;i<5;)
  {
  statement1;
  statement2;
  statement3;
  i++;
  }
```

however we have to explicitly alter the value i in the loop body.

Way 7 : Missing Initialization in For Loop

```
i = 0;
```

```
for(;i<5;i++)
  {
  statement1;
  statement2;
  statement3;
  }
```
we have to set value of 'i' before entering in the loop otherwise it will take garbage value of 'i'.

Way 8 : Infinite For Loop

```
i = 0;
for(;;)
  {
  statement1;
  statement2;
  statement3;

  if(breaking condition)
    break;

  i++;
  }
```

Infinite for loop must have breaking condition in order to break for loop. otherwise it will cause overflow of stack.

Summary of Different Ways of Implementing For Loop

Form	**Comment**

for (i=0 ; i < 10 ; i++)
Statement1;
 	
for (i=0 ;i <10; i++)
{
Statement1;
Statement2;
Statement3;
}
	Single Statement
	
	Multiple Statements within for

for (i=0 ; i < 10;i++) ;
 	
for
	For Loop with no Body (**Carefully Look at the Semicolon**)
	
	Multiple initialization & Multiple

(i=0,j=0;i<100;i++,j++) Statement1;	**Update** Statements Separated by Comma
for (; i<10 ; i++)	**Initialization not used**
for (; i<10 ;)	**Initialization & Update not used**
for (; ;)	**Infinite** Loop, Never Terminates

JUMP STATEMENTS:

Break statement
Break Statement Simply Terminate Loop and takes control out of the loop.

Break in For Loop :

```
for(initialization ; condition ; incrementation)
{
Statement1;
Statement2;
break;
}
```

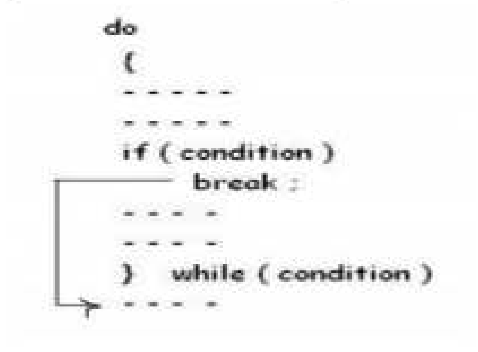
Break in While Loop :

```
initialization ;
while(condition)
{
Statement1;
Statement2;
incrementation
break;
}
```

Break Statement in Do-While :

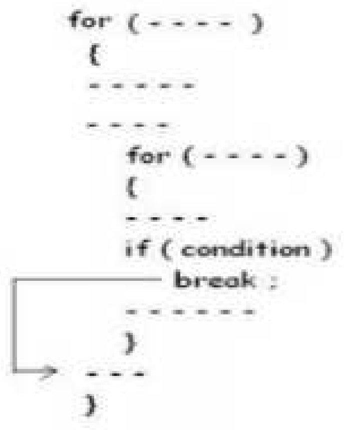
```
initialization ;
do
{
Statement1;
Statement2;
incrementation
break;
}while(condition);
```

Way 1 : Do-While Loop



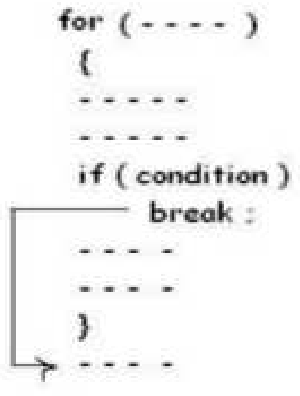
```
do
 {
 - - - - -
 - - - - -
 if ( condition )
      break ;
 - - - -
 - - - -
 }  while ( condition )
 - - - -
```

Way 2 : Nested for



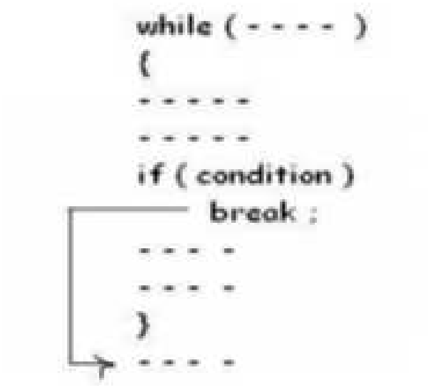
```
for ( - - - - )
 {
 - - - - -
 - - - -
    for ( - - - - )
    {
    - - - -
    if ( condition )
         break ;
    - - - - - -
    }
 - - -
 }
```

Way 3 : For Loop



```
for ( - - - - )
 {
 - - - - -
 - - - - -
 if ( condition )
      break ;
 - - - -
 - - - -
 }
 - - - -
```

Way 4 : While Loop



```
while ( - - - - )
{
- - - - -
- - - - -
if ( condition )
    break ;
- - - -
- - - -
}
- - - -
```

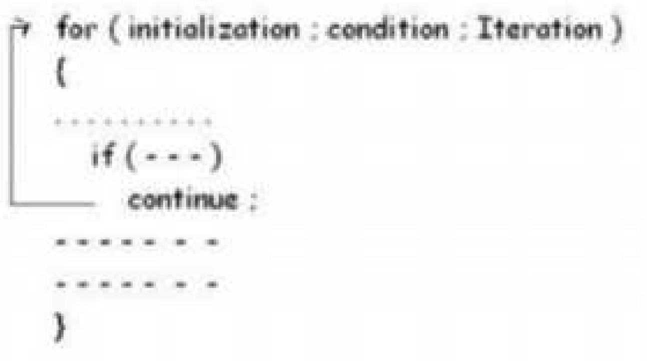
Continue statement:

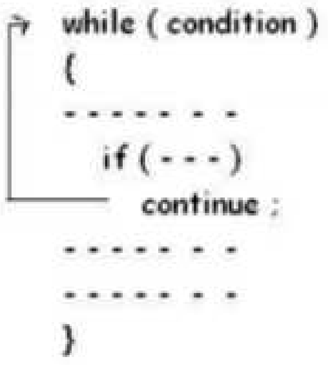
```
loop
{
 continue;
 //code
}
```

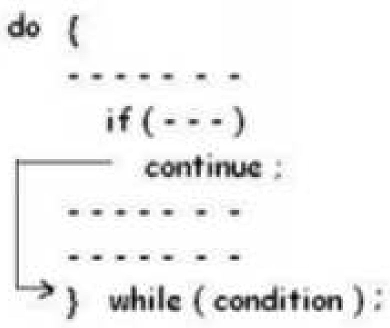
Note :

It is used for skipping part of Loop.

Continue causes the remaining code inside a loop block to be skipped and causes execution to jump to the top of the loop block

Loop	Use of Continue !!
for	 `for ( initialization ; condition ; Iteration ) { - - - - - - - - - - - if ( - - - ) continue ; - - - - - - - - - - - - - - }`

while	 while (condition)
	{
	- - - - - - -
	if (- - -)
	continue ;
	- - - - - - -
	- - - - - - -
	}

do-while	 do { - - - - - - - if (- - -) continue ; - - - - - - - - - - - - - - } while (condition) ;

Goto statement:

goto label;

----

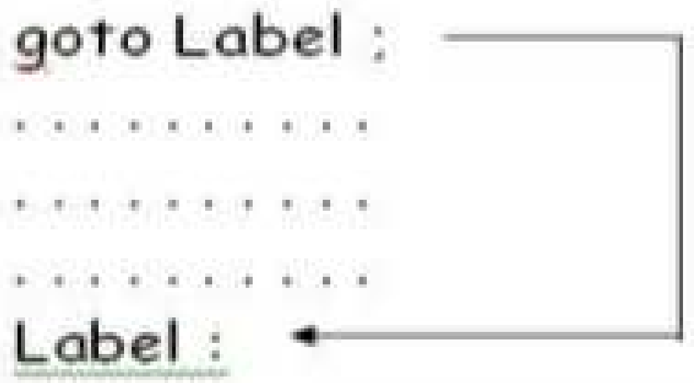
label :

Whenever goto keyword encountered then it causes the program to continue on the line , so long as it is in the scope .

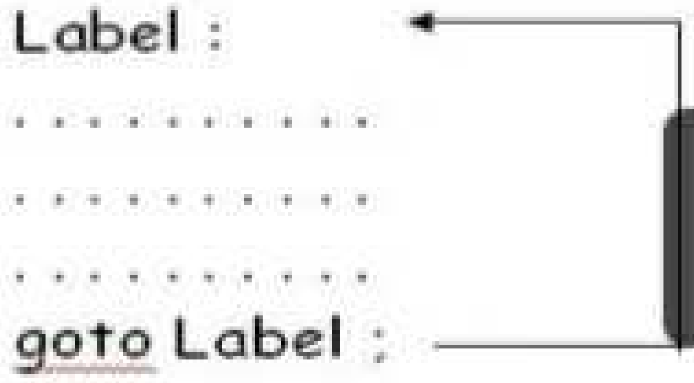
<u>Types of Goto</u>

Forward

Backward



```
goto Label ;
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
Label :
```



```
Label :
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
goto Label ;
```

71 of 171