

SUBJECT TITLE AND CODE : INTRODUCTION TO C PROGRAMMING – 1BPLC105E

MODULE 2

OPERATORS AND EXPRESSIONS

C support a rich set of built-in operators. An operator is a symbol that tells the compiler to perform specific mathematical or logical functions. C language is rich in built-in operators and provides the following types of operators:

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Assignment Operators
5. Increment and decrement Operators
6. Conditional Operators
7. Bitwise Operators
8. Special Operators

Arithmetic Operators

C provides all the basic arithmetic operators. The operator +, -, * and / all work same way as in other languages. These can operate on any built-in data type allowed in C.

Table 3.1 Arithmetic Operators

Operator	Meaning
+	Addition or unary plus
-	Subtraction or unary minus
*	Multiplication
/	Division
%	Modulo division

Integer Arithmetic

When both the operands in a single arithmetic expression such as $a+b$ are integers, the expression is called the integer expression, and the operation is called as integer arithmetic. Integer arithmetic always results in an integer value. The largest integer value depends on the machine.

Example a and b are integers, if $a=14$ and $b=4$ then,

$a - b = 10$, $a + b = 18$, $a * b = 56$

$a / b = 3$ (decimal part truncated), $a \% b = 2$ (remainder of division)

During integer division, if both the operands are of the same, the result is truncated towards zero. If one of them is negative then the direction of truncation is implementation dependent.

$6/7=0$ and $-6/-7=0$

But $-6/7$ may be 0 or -1

During modulo division, the sign of the result is always the sign of the first operand (the dividend).

$-14 \% 3 = -2$

$-14 \% -3 = -2$

$14 \% -3 = 2$

Real Arithmetic

An arithmetic involving only real operands is called real arithmetic. A real operand may assume values either in decimal notation or in exponent notation. Since floating point values are rounded to the number of significant digits permissible, the final value is an approximation of the correct result.

Example x,y and z are floats $x=6.0/7.0 = 0.857143$

$y=1.0/3.0 = 0.333333$

$z=-2.0/3.0= -0.666667$

The operator modulo (%) cannot be used with real operands.

Mixed-mode Arithmetic

When one operand is real and the other is integer, the expression is called a mixed-mode arithmetic expression. If either operand is of real type, then only the real operation is performed and the result is always a real number.

Example

$15 / 10.0 = 1.5$

where as,

$15 / 10 = 1$

Relational Operators

We often compare two quantities and depending on their relation, take certain decisions. These comparisons can be done with the help of relational operators.

Example

$a < b$ or $1 < 20$

containing a relational operator is termed as a relational expression. The value of relational expression is either true (non-zero) or false (zero) value.

$10 > 20$ is true

$20 < 10$ is false

C supports six relational operators

Table 3.2 Relational Operators

Operator	Meaning
<	is less than
<=	is less than or equal to
>	is greater than
>=	is greater than or equal to
==	is equal to
!=	is not equal to

Syntax:

ae-1 relational operator ae-2

ae-1 and ae-2 are arithmetic expressions, which may be simple constants, variables are combination of them.

4.5 <= 10 TRUE

4.5 < -10 FALSE

10 < 7+5 TRUE

a+b=c+d TRUE only if the sum of a and b is equal to the sum of c and d.

When arithmetic expressions are used on either side of a relational operator, the arithmetic operators have a higher priority over relational operators.

Relational Operator Complements

Among the six relational operators, each one is a complement of another operator.

>	is complement of	<=
<	is complement of	>=
==	is complement of	!=

We can simplify an expression involving the *not* and the *less than* operators using the complements as shown below:

Actual one	Simplified one
!(x < y)	x >= y
!(x > y)	x <= y
!(x != y)	x == y
!(x <= y)	x > y
!(x >= y)	x < y
!(x == y)	x != y

Logical Operators

C has the following 3 logical operators.

&&	logical AND
 	logical OR
!	logical NOT

Logical Operators are used when we want to test more than one condition and make decisions. It combines two or more relational expressions, is termed as a logical expression or compound relational expression.

Example

`a > b && x == 10`

This expression is true only if `a > b` is true and `x == 10` is true

Table 3.3 Truth Table			
op-1	op-2	Value of the expression	
		<code>op-1 && op-2</code>	<code>op-1 op-2</code>
Non-zero	Non-zero	1	1
Non-zero	0	0	1
0	Non-zero	0	1
0	0	0	0

Example

`if (age > 50 && salary < 1000) if (number < 0 || number > 100)`

Assignment operators

Assignment operators are used to assign the result of an expression to a variable. Assignment operator is '='.

Syntax

`v op = exp; or v = v op (exp);`

where `v` is a variable, `exp` is an expression and `op` is a C binary arithmetic operator. The operator `op=` is known as the shorthand assignment operator. Example

`x += y + 1;`

This is same as

`x = x + (y + 1);`

The shorthand operator `+=` means 'add `y+1` to `x`' or increment `x` by `y+1`. For `x=5` and `y=2` means

`x = 5 + (2 + 1);`

`x = 5 + 3; x = 8;`

Simple Statements	Assignment Statements	Operators
<code>a=a+1 a=a-1</code> <code>a = a * (n+1) a = a / (n+1)</code> <code>a=a%b</code>	<code>a +=1</code> <code>a -=1</code> <code>a *= n + 1 a /= n + 1 a %= b</code>	

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

```

Program
#define N 100
#define A 2
main()
{
    int a;
    a = A;
    while( a < N )
    {
        printf("%d\n", a);
        a *= a;
    }
}

Output
2
4
16

```

Increment and decrement operators

- The **increment operator** (++) increases the value of a variable by **1**.
- The **decrement operator** (--) decreases the value of a variable by **1**.

Examples:

```

++m;    m++;
--m;    m--;

```

Equivalent forms:

```
++m → m = m + 1 or m += 1  
--m → m = m - 1 or m -= 1
```

These operators are very commonly used in loops and expressions.

Prefix and Postfix Forms

The operators ++ and -- can be used in **two ways**:

1. Prefix Form (++m / --m)

- The variable is **incremented or decremented first**.
- Then the **updated value** is used in the expression.

2. Postfix Form (m++ / m--)

- The **current (old) value** is used first in the expression.
- The variable is updated **after** the expression is evaluated.

Example to Understand the Difference

```
m = 5;  
y = m++;
```

- Postfix is used → so **y gets the old value 5**
- Then **m becomes 6**

But,

```
m = 5;  
y = ++m;
```

- Prefix is used → **m becomes 6 first**
- Then **y also becomes 6**

Using ++ and -- in Complex Expressions

Increment and decrement operators can also appear inside larger expressions.

Example:

```
m = n++ - j + 10;
```

- Here **n** is incremented only after the whole expression is evaluated.

Important Rules

1. ++ and -- are **unary operators**, meaning they work on a single variable.
2. **Postfix** (++ / --) → use the old value first, then increment/decrement.
3. **Prefix** (++ / --) → increment/decrement first, then use the new value.
4. Their **precedence and associativity** are the same as unary + and unary −.
5. These operators work **only with variables**, not constants.

Conditional operator

A ternary operator “?:” is available in C to construct expressions.

SYNTAX

exp1?exp2:exp3; where exp1,exp2 and exp3 are expressions.

exp1 is evaluated first. If it is non-zero (true) then, exp2 is evaluated and becomes the value of the expression.

If exp1 is false, exp3 is evaluated and becomes the value of the expression.

```
a=10; b=15;
x = (a>b) ? a : b ; In Simple way,
if (a>b)
x = a;
else
x = b;
```

Bitwise operators

C has a distinction of supporting special operators known as bitwise operators for manipulation of data at bit level. These operators are used for testing the bits, or shifting them right or left. Bitwise may not be applied to float or double.

Operator	Meaning
&	Bitwise AND
	Bitwise OR
^	Bitwise XOR
<<	Shift left
>>	Shift right
~	One's complement

Special operators

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

Comma Operator

Comma-linked list of expressions are evaluated left to right...and the value of right most expression is the value of the combined expression. Example

value = (x=10, y=5, x+y);

x will get 10 then y will get 5 then expression x+y is evaluated as 15.

sizeof operator

The sizeof is a compile time operator and, when used with operand, it returns the no. of bytes the operand occupies. The operand may be a variable, a constant or a data type qualifier.

The sizeof operator normally used to determines the length of entities, arrays and structures when their sizes are not known to the programmer. It is also use to allocate size of memory dynamically during execution of the program.

Example

```
main( )
{
    int sum; float f;
    printf( "The size of f %d\n The size of sum%d" ,size of(f), size of (sum) );
}
```

Output

The size of f 4 The size of sum 2

Arithmetic expression

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

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

Evaluation of Expressions

Expressions are evaluated using an assignment statement of the form

Variable=expression;

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.

$x=a*b-c$; $y=b/c*a$; $z=a-b/c+d$;

Precedence of arithmetic operators

An arithmetic operation without parentheses will be evaluated from left to right using the rules of precedence of operators.

High priority $*$ / $\%$ Low priority $+$ $-$

The basic evaluation includes left to right passes. During first pass higher priority operators are applied as they encountered. During second pass from left to right the low priority operators are applied as they are encountered.

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

when $a=9$, $b=12$ and $c=3$, the statement becomes

$x=9-12/3+3*2-1$

and is evaluated as follows

$x = 9-12/3+3*2-1$;

pass 1:

$x = 9 - 4 + 3*2 - 1$;

$x = 9 - 4 + 6 - 1$;

Pass 2:

$x = 5 + 6 - 1$; $x=11-1$; $x=10$;

Thus, when parentheses are used, the expressions within parenthesis assume highest priority.

$x = 9 - 12 / (3+3) * (2-1)$

Pass 0:

$x=9 - 12 / 6 * (2-1)$

$x=9 - 12 / 6 * 1$;

Pass 1:

$x=9 - 2 * 1$;

$x=9 - 2$;

Pass 2:

$x=7$;

Parentheses may be nested, and in such cases, evaluation of the expression will proceed outward from the innermost set of parentheses.

Example

$x = 9 - (12 / (3+3) *)2-1=4$

Rules

1. First, parenthesized sub expressions from left to right are 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 two or more operators of the same precedence level appear in the 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.

Decision making, Branching and Looping

C program is a set of statements which are normally executed sequentially in the order in which they appear. We have a number of situations where we may have to change the order of execution of statements based on the certain conditions or repeat a group of statements until certain specified conditions are met.

C Language supports the control statements that are;

- a) If Statements
- b) If-else statements
- c) Else-if ladder Statements
- d) Switch-case statements

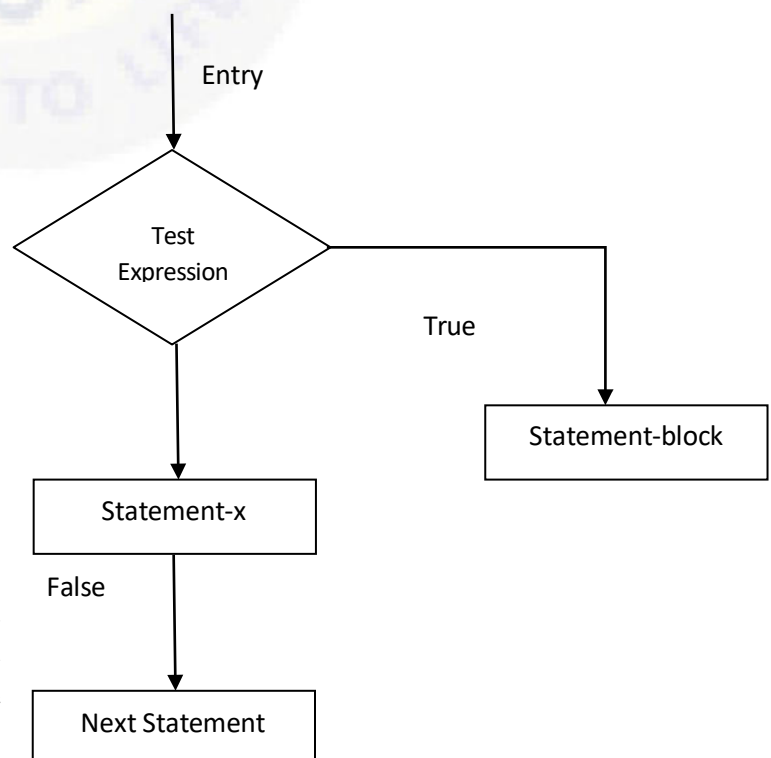
If Statements

The general form of a simple if statement is

Syntax

```
if(test expression)
{
Statement-block;
}
```

The 'statement-block' may be a single statement or a group of statement. If the test expression is true, the statement-block will be executed; otherwise the statement-block will be



skipped.

Example

```
void main()
{
    int n;

    printf("enter a number:"); scanf("%d",&n);
    if (n>10)
        Printf(" number is grater");
}
```

Output:

Enter a number:12

Number is greater

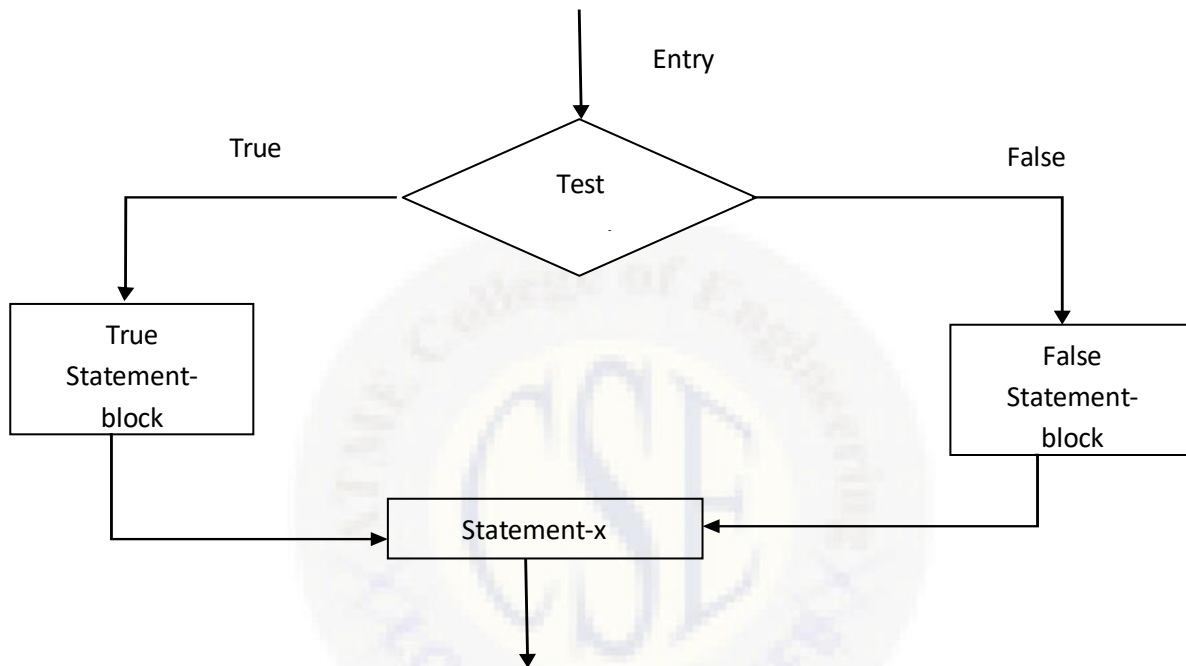
if – Else Statements

The if...else statement is the extension of simple if statement.

Syntax:

```
if(test expression)
{
    True-block Statement;
}
else
{
    False-block Statement;
}
```

If the test expression is true, then the true-block statements are executed. Otherwise the false-block statements are executed. In either case, either true-block or false-block will be executed, not both.



Example

```
void main()
{
    int n;
    printf("enter a number:");

    scanf("%d", &n);

    if (n%2==0)
        printf("even number");

    else
        printf("odd number");
}
```

```
}
```

Output:

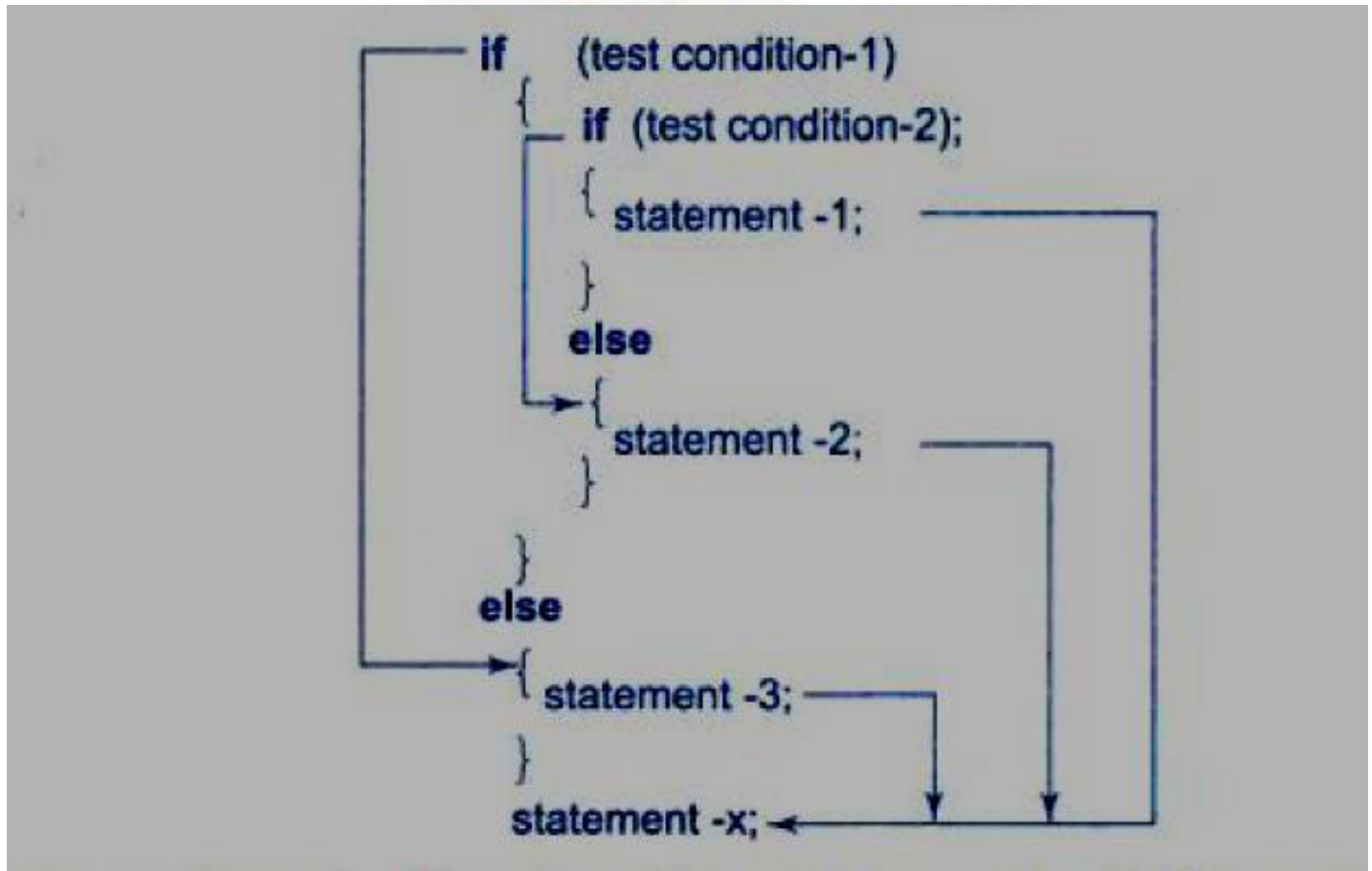
enter a number:121

odd number

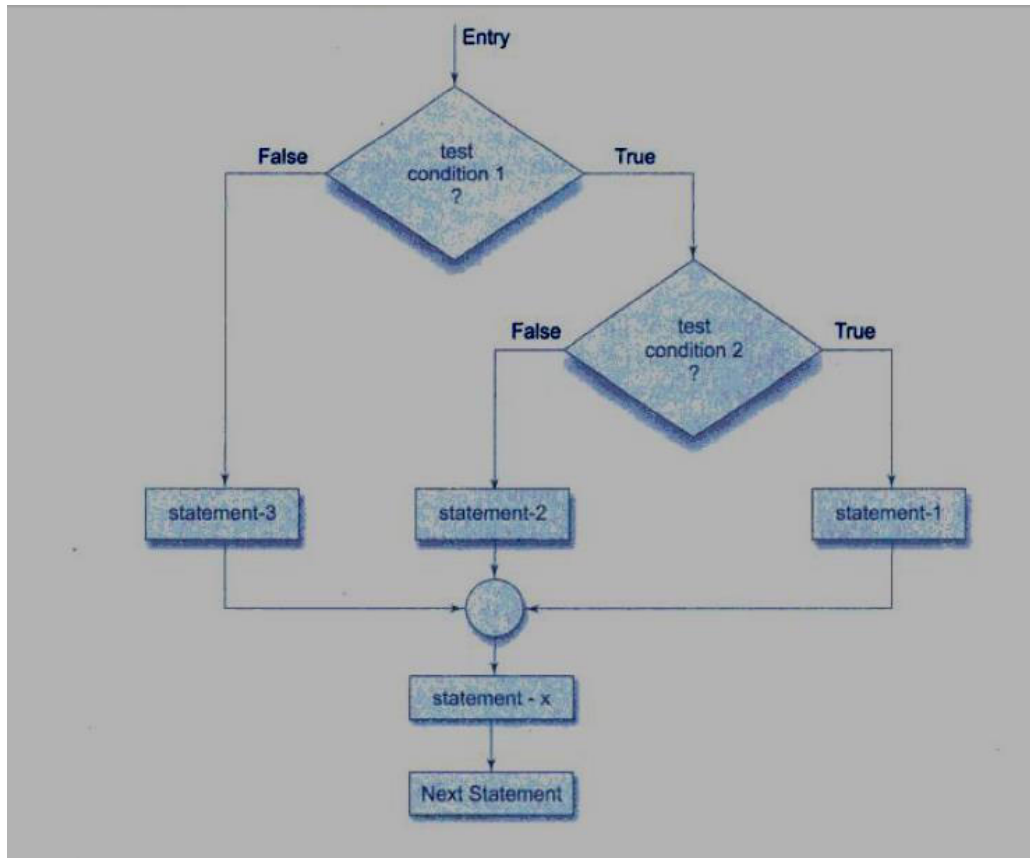
Nested If Statements

When a series of decisions are involved, we may have to use more than one if....else

Syntax



If the condition-1 is false, the statement-3 will be executed; otherwise it continues to perform the second test. If the condition-2 is true, statement-1 will be executed; otherwise statement-2 will be executed and the control is transferred to the statement-x.



Example

```
#include<stdio.h> #include<conio.h> int main()
{
    int a,b,c;
    printf("Enter three integers to check\n"); scanf("%d %d %d",&a,&b,&c);
    if(a>b)
    {
        if(a>c)
        {
            printf("a is big");
        }
    }
}
```

```
}  
else  
{  
printf("c is big");  
}  
}  
else  
{  
if(b>c)  
{  
printf("b is big");  
}  
else  
{  
printf("c is big");  
}  
}  
getch();  
}
```

Output

Enter three integers to check 33 67 2

b is big

Else - If Ladder

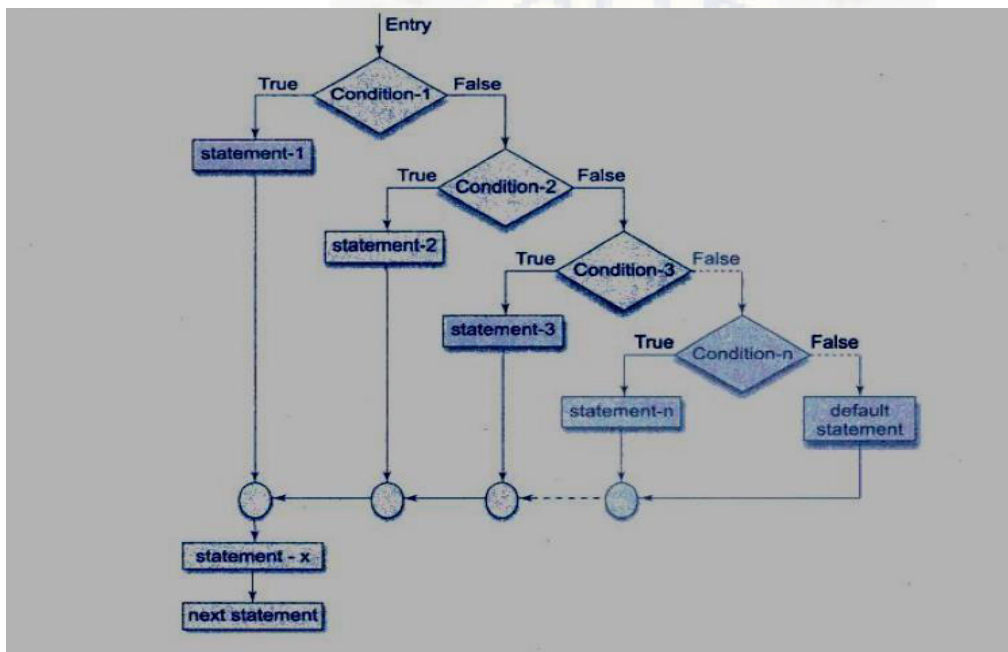
When multiple decisions are involved, chain of if statement associated with each else. This construct is known as else... if ladder.

General format is;

```
if(condition 1)
```

```
statement1;
```

else if (condition2)
statement 2;
else if (condition 3)
statement 3;
else if (condition n)
statement n;
else
Default-statement
statement x;



The conditions are evaluated from the top downwards. As soon as a true condition is found, the statement associated with it is executed with it is executed and the control is transferred to the statement-x. When all the n conditions become false, then the final else containing the default-statement will be executed.

Example

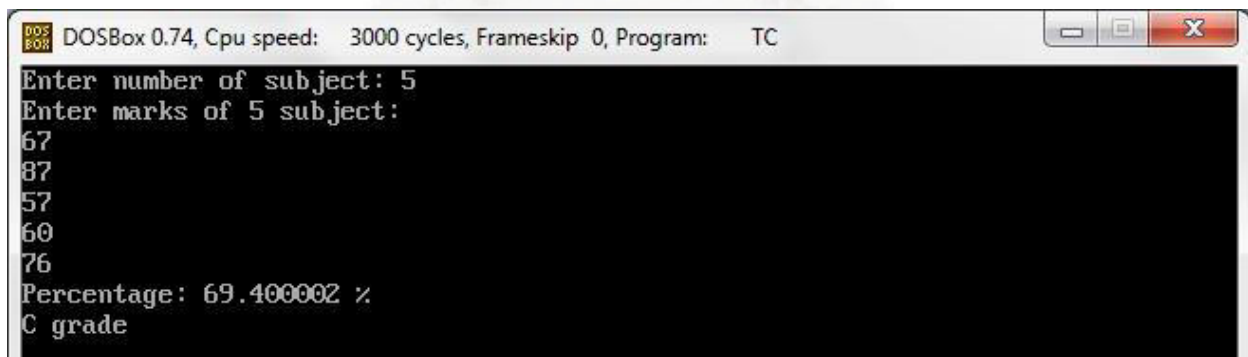
```
#include<stdio.h>

#include<conio.h>

void main()
{
    int n, i;
    float marks[10], per=0, total=0;
    clrscr();
    printf("Enter number of subject: ");
    scanf("%d",&n);
    printf("Enter marks of %d subject: ",n);
    for(i=0; i<n; i++)
    {
        scanf("%f",&marks[i]);
        total=total+marks[i];
    }
    per=total/n;
    printf("Percentage: %f % \n",per);
    if(per>=85)
    {
        printf("A grade");
    }
    else if(per>=75 && per<85)
    {
        printf("B grade");
    }
    else if(per>=50 && per<75)
    {
        printf("C grade");
    }
    else if(per>=35 && per<50)
```

```
{  
printf("D grade");  
}  
else  
{  
printf("Fail");  
}  
getch();  
}
```

OUTPUT

A screenshot of a DOSBox window titled "DOSBox 0.74, Cpu speed: 3000 cycles, Frameskip 0, Program: TC". The window has a black background with white text. The text shows the program's execution: "Enter number of subject: 5", "Enter marks of 5 subject:", followed by five lines of input marks (67, 87, 57, 60, 76), then "Percentage: 69.400002 %", and finally "C grade".

```
DOSBox 0.74, Cpu speed: 3000 cycles, Frameskip 0, Program: TC  
Enter number of subject: 5  
Enter marks of 5 subject:  
67  
87  
57  
60  
76  
Percentage: 69.400002 %  
C grade
```

Switch statement

Switch statement is used for complex programs when the number of alternatives increases. The switch statement tests the value of the given variable against the list of case values and when a match is found, a block of statements associated with that case is executed.

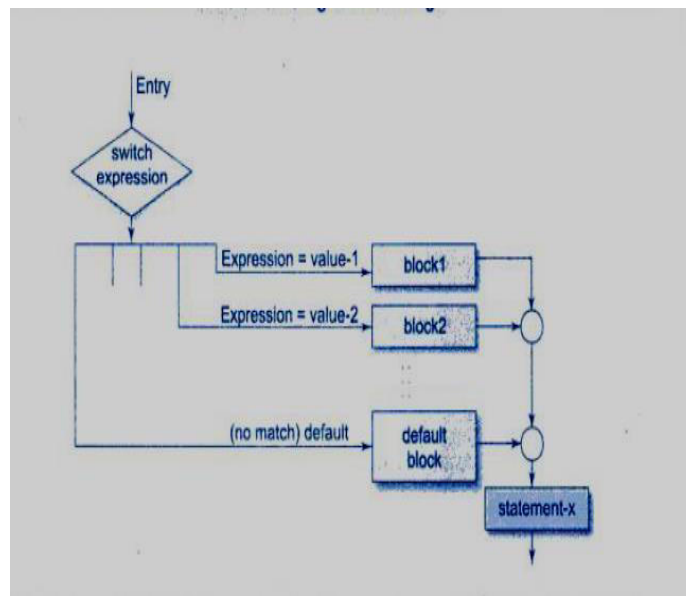
Syntax

The general form of switch statement is

```
switch(expression)  
{  
case value-1: block-1; break;  
case value-2: block-2;  
break;
```

.....

```
..... default:  
default-block; break;  
}  
statement- x;
```



Rules

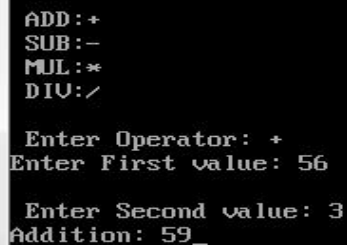
1. The expression is an integer expression or characters.
2. Value-1, value-2... are constants or constant expressions are known as case labels.
3. These values are unique within switch statement.
4. Block-1, block-2.... are statement lists and may contain zero or more statements.
5. Case labels end with a colon (:)
6. When the switch is executed, the value of the expression is successfully compared against the values value-1, value-2,... If a case is found whose value matches with the value of the expression, then the block of statements that follows the case are executed.
7. The break statement transfers the control out of the switch statement. That is two or more case labels may belong to the same statements.
8. The default is an optional case. When present, it will be executed if the value of the expression does not match with any of the case values. If not present, no action takes place if all matches fail and the control goes to the statement-x.

Example

```
#include<stdio.h> #include<conio.h> void main()
{
char choice; int a,b,res=0; clrscr();
printf("\n ADD:+\n SUB:-\n MUL:*\n DIV:/\n"); printf("\n Enter Operator: ");
scanf("%c",&choice);
printf("Enter First value: "); scanf("%d",&a);
printf("\n Enter Second value: "); scanf("%d",&b);
switch(choice)
{
case '+':
res=a+b;
printf("Addition: %d",res); break;
case '-':
res=a-b;
printf("Subtraction: %d",res); break;
case '*':
res=a*b;
printf("Multiplication: %d",res); break;
case '/':
res=a/b;
printf("Division: %d",res); break;
default:
printf("Enter Valid Operator!!");
}

getch();
}
```

OUTPUT



```
ADD: +
SUB: -
MUL: *
DIV: /

Enter Operator: +
Enter First value: 56

Enter Second value: 3
Addition: 59_
```

The ?: operator

The operator is a combination of ? and : and takes three operands. This operator is popularly known as the conditional operator.

Syntax

Conditional `expression? expression1:expression2`

The conditional expression is evaluated first. If the result is non-zero expression1 is evaluated and is returned as the value of the conditional expression. Otherwise, expression2 is evaluated and its value is returned.

Example

```
#include <stdio.h> int main()
{
    int num;
    printf("Enter the value ");
    scanf("%d",&num);
    ((num%2)==0) ? printf("%d is even.",num) : printf("%d is odd.",num); return 0;
}
```

Output

Enter the value 5

5 is odd

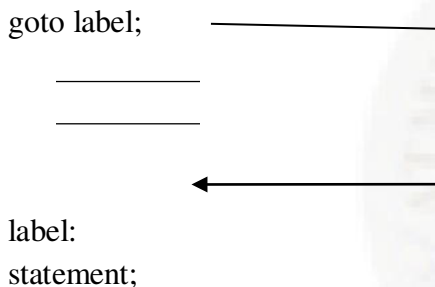
THE GOTO STATEMENT

C supports the goto statement to branch unconditionally from one point of the program to another. The goto requires a label in order to identify the place where the branch is to be made. A label is any valid variable name and must be followed by a colon. The label is placed immediately before the statement where the control is to be transferred.

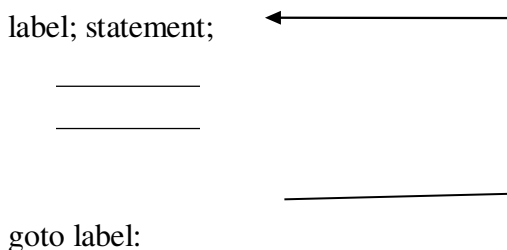
Syntax

The general form is

Forward jump: If the label: is placed after the goto label; some statements will be skipped and the jump is known as a forward jump.



Backward Jump: If the label: is placed before the goto label; a loop will be formed and some statements will be executed repeatedly and the jump is known as a backward jump.



Example:

```
main()  
{  
  
double x,y; read; scanf("%f",&x);  
if(x<0) goto read;  
y=sqrt(x);
```

```
printf("%f%f",x,y); goto read;  
}
```

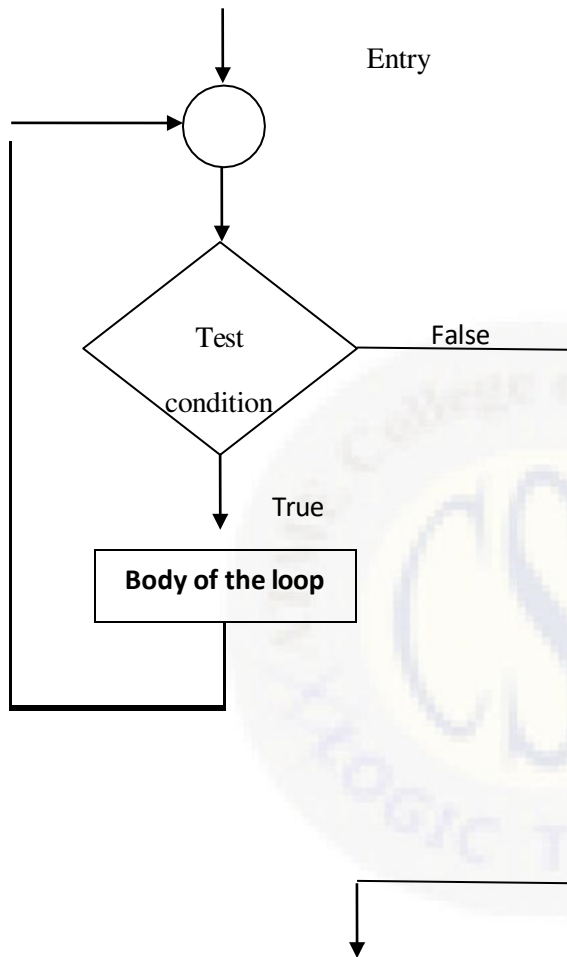
DECISION MAKING AND LOOPING

It is possible to execute a segment of a program repeatedly by introducing a counter and later testing it using the if statement. While this method is quite satisfactory for all practical purposes, we need to initialize and increment a counter and test its value at an appropriate place in the program for the completion of the loop.

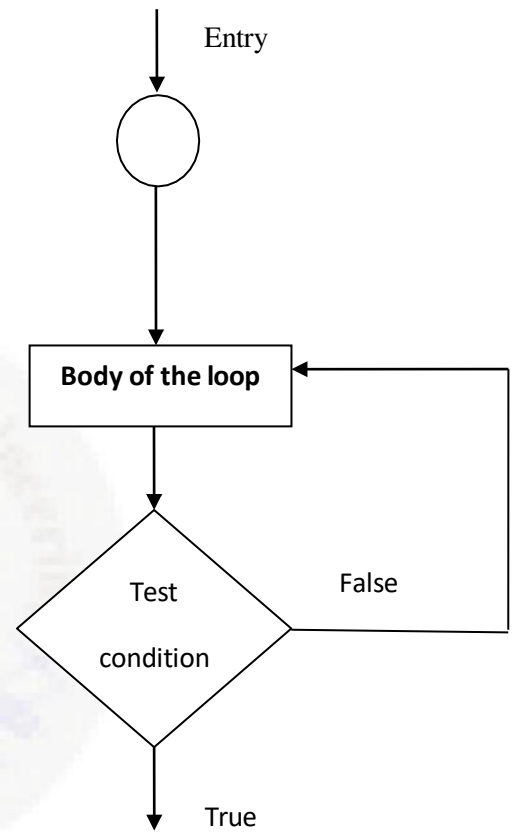
In looping, a sequence of statements is executed until some conditions for termination of the loop are satisfied. A program loop therefore consists of two segments, one known as the body of the loop and the other known as the control statements.

Depending on the position of the control statements in the loop, a control structure may be classified either as an entry-controlled loop or as the exit-controlled loop. **Entry-controlled loop:** The control statements are tested before the start of the loop execution. If the conditions are not satisfied, then the body of the loop will not be executed.

Exit-controlled loop: The test is performed at the end of the loop and therefore the body is executed unconditionally for the first time



Entry Controlled loop



Exit Controlled

The while statement

The while is an entry-controlled loop statement. The test-condition is evaluated and if the condition is true, then the body of the loop is executed. After execution of the body, the test-condition is once again evaluated and if it is true, the body is executed once again. This process of repeated execution of the body continues until the test-condition finally becomes false and the control is transferred out of the loop.

Syntax

```
while(test condition)
{
    body of the loop;
}
```

Example

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int s=0,a,n; clrscr();
    printf("Enter the number"); scanf("%d",&n); while(n>0)
    { a=n%10;
      s=s+a; n=n/10;
    }
    printf("\nSum of digits: %d\t",s); getch();
}
```

Output

```
Enter the number
436

Sum of digits: 13  _
```

The do statement

In while loop the body of the loop may not be executed at all if the condition is not satisfied at the very first attempt. On some occasions it might be necessary to execute the body of the loop before the test is performed. Such situations can be handled with the help of the do statement.

Syntax

```
do{  
body of the loop  
}  
while(test condition);
```

The program proceeds to evaluate the body of the loop first. At the end of the loop, the test condition in the while statement is evaluated. If the condition is true, the program continues to evaluate the body of the loop once again. This process continues as long as the condition is true. When the condition becomes false, the loop will be terminated and control goes to after the while statement.

Since the test-condition is evaluated at the bottom of the loop, the do.....while construct provides an exit-controlled loop and therefore the body of the loop is always executed at least once.

Example

```
#include<stdio.h>  
void main()  
{  
int i=1;  
do  
{  
printf("%d\n",i);  
i++;  
}  
while(i<=5);  
}
```

Output

1
2
3
4
5

The for statement

The for loop is another entry-controlled loop that provides a more consistent loop control structure.

Syntax

The general form of the for loop is:

```
for(initialization ; test-condition ; increment)
```

```
{
```

```
body of the loop
```

```
}
```

1. Initialization of the control variables is done first.
2. The value of the control variable is tested using the test-condition. If the condition is true, the body of the loop is executed; otherwise the loop is terminated and the execution continues with the statement that immediately follows the loop.
3. When the body of the loop is executed, the control is transferred back to the for statement after evaluating the last statement in the loop. Now, the control variable is either incremented or decremented as per the condition.

Example

```
#include<stdio.h>

#include<conio.h>

void main()
{
    int i,n,a = -1,b= 1,c;
    clrscr();
    printf("Enter the number\t");
    scanf("%d",&n);
    printf("\n Fibonacci Series \n")
    for(i=0;i<n;i++)
    {
        c=a+b;
        printf("%d\n",c);
        a=b;
        b=c;
    }
    getch();
}
```

Output

```
Enter the number      5

Fibonacci Series
0
1
1
2
3
```

Additional feature of the loop

- More than one variable can be initialized at a time and it can be separated by comma.

`for(i=0,n=0;n<17;n++)`

- The multiple arguments in the increment section are possible and separated by commas.

`for(n=1,m=50;n<=m;n++,m--)`

- The test condition may have any compound relation and the testing need not be limited only to the loop control variable.

`for(i=1;i<20 && sum<100;i++)`

- To use expressions in the assignment statements of initialization and increment sections

`for(x=(m+n)/2;x>0;x=x/2)`

Nesting of for loops

Nesting of loops, that is, one for statement within another for statement, is allowed in C. `for(i=1;i<10;i++)`

```
{  
  for(j=1;j!=5;j++)
```

```
{  
    -----  
    -----  
}
```

```
    -----  
    -----  
}
```

Example

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a[50],i,j,n,temp;
    clrscr();
    printf("Enter the array boundary\n");
    scanf("%d",&n);
    printf("Enter %d array element\n",n);
    for(i=1;i<=n;i++)
        scanf("%d",&a[i]);
    for(i=1;i<=n;i++)
    {
        for(j=i+1;j<=n;j++)
        {
            if(a[i]>a[j])
            {
                temp=a[i];
                a[i]=a[j];
                a[j]=temp;
            }
        }
    }

    printf("Sorted Array\n");
    for(i=1;i<=n;i++)
        printf("%d\n",a[i]);
    getch();
}
```



OUTPUT

```
Enter the array boundary
5
Enter 5 array element
45
67
3
90
2
Sorted array
2
3
45
67
90
_
```