MODULE 2

Operators in C, Type conversion and typecasting

Decision control and Looping statements: Introduction to decision control, Conditional branching statements, iterative statements, nested loops, break and continue statements, goto statement.

Textbook: Chapter 9.15-9.16, 10.1-10.6

Chapter 2: Decision control and looping statements:

- A **Statement** causes an action to be performed.
- A statement is a group of tokens ended with a semicolon.
- A C program is a set of statements which are normally executed sequentially in the order in which they appear.
- However, in practice there are number of situations where we may have to change the order of execution of statements based on conditions, or repeat a group of statements until certain specified conditions are met.

So C, possesses decision making capabilities – These statements are popularly known as decision making statements. Also known as control statements or as control structures.

Categories of control structures:

Decision making and branching (conditional statements) – specifies a condition to execute one or more statements in a program. (if, switch, conditional operator)

Decision making and branching (Unconditional statements) – changes the normal sequence of the program execution by transferring control to other part of the program (goto, continue, break)

Decision making and Looping (Iterative statements) – executes one or more statements repeatedly until some condition is met. (while, do while, for)

Conditional branching statements

C language possesses such decision making capabilities and branching (conditional statements) by supporting the following statements

1. **if** statement
2. **switch** statement
3. Conditional operator statement

DECISION MAKING WITH IF STATEMENT

The if statement is a **two way decision statement**.

It allows the computer to evaluate the expression first and then, depending on whether the value of the

expression (relation or condition) is 'true' (non zero) or 'false' (zero), it transfers the control to a particular statement. This point of program has two paths to follow, one for the true condition and the other for the false condition.

The if statement may be implemented in different forms depending on the complexity of conditions to be tested. They are:

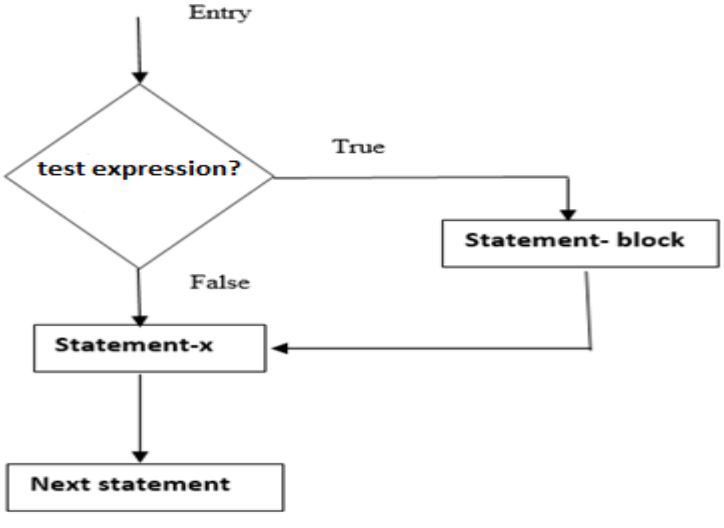
1. Simple **if** statement
2. **If….else** statement
3. **Nested if …..else** statement
4. **if else if** ladder

SIMPLE IF STATEMENT

The general form of a simple if statement is:

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

The 'statement-block' may be a single statement or a group of statements. If the test expression is true, the statement-block will be executed; otherwise the statement-block will be skipped and the execution will jump to the statement-x. When condition is true both the statement-block and statement-x are executed in sequence.



Program: eligible to vote or not

```
#include<stdio.h>
```

```
int main()
  {
        int age;
        printf("\n Enter age ");
        scanf("%d ", &age);
        if(age>=18)
              printf("\n Eligible to vote");
        if(age<18)
              printf("\n Not eligible to vote");
        return 0;
}
```

In case the statement block contains only one statement, putting curly brackets becomes optional. If there is more than one statement in the statement block, putting curly brackets becomes mandatory.

THE IF….ELSE STATEMENT

The if…..else statement is an extension of the simple if statement. The general form is:

```
if (test expression)
{
      true-block statement(s);
}
else
{
      false-block statement(s);
}
Statement-x;
```

If the test expression is true, then the true-block statement(s), immediately following the if statement are executed; otherwise, the false-block statement(s), are executed. In either case true-block or false-block will be executed, not both.

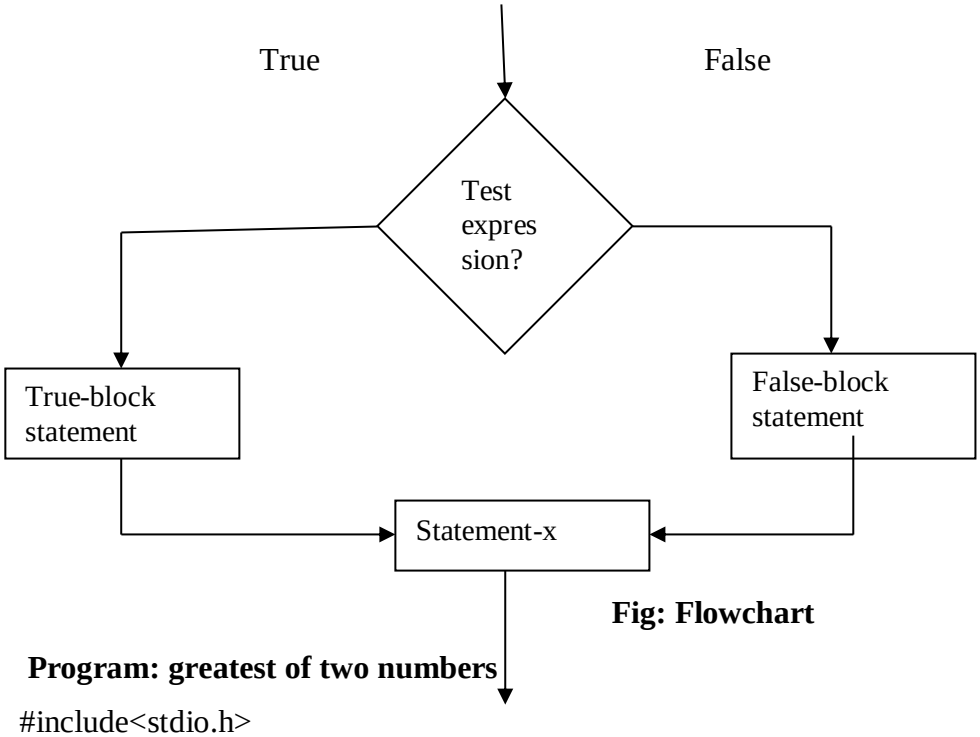


Fig: Flowchart

Program: greatest of two numbers

```
#include<stdio.h>
int main()
{
        int a, b;
        printf("\n Enter two integer values");
        scanf("%d %d ", &a, &b);
        if(a>b)
                printf("%d is greatest",a);
        else
                printf("%d is greatest",b);
        return 0;
}
```

Program: odd or even

```
#include<stdio.h>
int main()
{
        int a;
        printf("Enter a value ");
        scanf("%d ", &a);
        if(a%2==0)
```

```
        {
                printf("%d is even",a);
        }
        else
        {
                printf("%d is odd",a);
        }
        return 0;
}
```

Program: eligible to vote or not

```
int main()
  {
        int age;
        printf("Enter age ");
        scanf("%d ", &age);
        if(age>=18)
                printf("eligible to vote");
        else
                printf("not eligible to vote");
        return 0;
}
```

Program: vowel or not

```
#include<stdio.h>
#include<ctype.h>
int main()
{
        char c;
        printf("Enter a valid character ");
        scanf("%c", &c);
        c=tolower(c);
        if(c=='a'||c=='e'||c=='i'||c=='o'||c=='u')
                printf("Input character is a vowel");
        else
                printf("Input character is a consonant");
```

```
    return 0;
}
```

Program: leap year or not

```
#include<stdio.h>
int main()
{
    int year;
    printf("Enter a valid year ");
    scanf("%d", &year);
    if((year%4==0&&year%100!=0)|| year%400==0)
            printf("Input year is a leap year");
    else
            printf("Input year is not a leap year");
    return 0;
}
```

NESTING OF IF…ELSE STATEMENTS

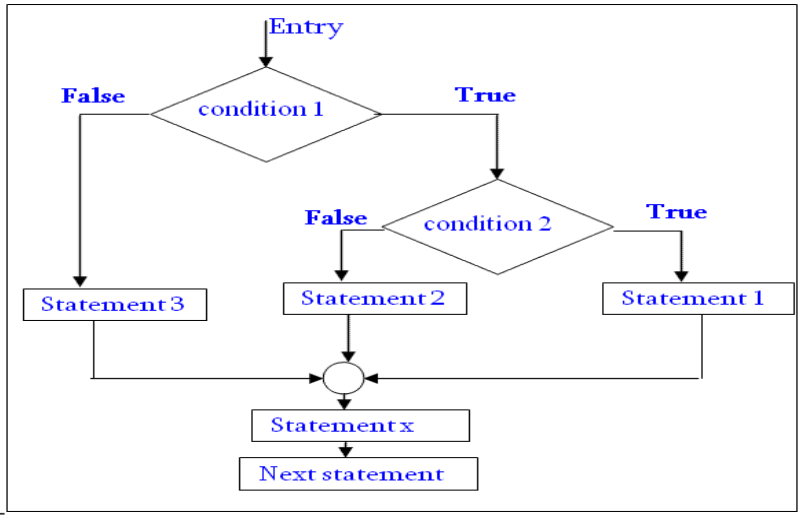
When a series of decisions are involved, we may have to use more than one **if…else** statements in nested form as shown below

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 the statement-1 will be evaluated; otherwise the statement-2 will be evaluated and then the control is transferred to the statement-x.

Syntax:

```
if (test condition - 1)
  {
  if (test condition - 2)
    {
      statement 1;
    }
   else
    {
      statement 2;
    }
  }
else
  {
   statement 3;
  }
 statement x;
```

Fig: Flowchart



Program

```
#include<stdio.h>
int main()
{
float  a,b,c;
printf("Enter three values\n");
scanf("%f %f %f", &a, &b, &c);
if (a>b)
{
        if (a>c)
                printf("\n %f is largest", a);
        else
                printf("%f is largest ", c);
}
else
{
        if (c>b)
                printf("%f is largest ",c);
        else
                printf("%f is largest ", b);
  }
   return 0;
  }
```

Output

Enter three values

23445 67379 88843

Largest value is 88843.000000

DANGLING ELSE PROBLEM

One of the classic problems encountered when we start using nested **if….else** statement is the dangling else. This occurs when a matching **else** is not available for an **if**. The answer to this problem is very simple.

Always match an **else** to the most recent unmatched **if** in the current block. In some cases, it is possible that the false condition is not required. In such situations, **else** statement may be omitted. (**else** is always paired with **recent unpaired if** in the current block)

```
        if(n>0)
        if(a>b)
        z=a;
        else
```

```
z=b;
```

The else is associated with if(a>b). If we want else to be associated with if (n>0), then brackets should be used like this:

```
if(n>0)
{
if(a>b)
z=a;
}
else
z=b;
```

THE ELSE IF LADDER

Nested if else is difficult to understand and modify. So we use else if ladder. Also called cascaded if else. There is another way of putting **if**s together when multipath decision are involved. A multipath decision is a chain of **if**s in which the statement associated with each **else** is an **if.** This construct is known as the **else if** ladder.

Syntax:

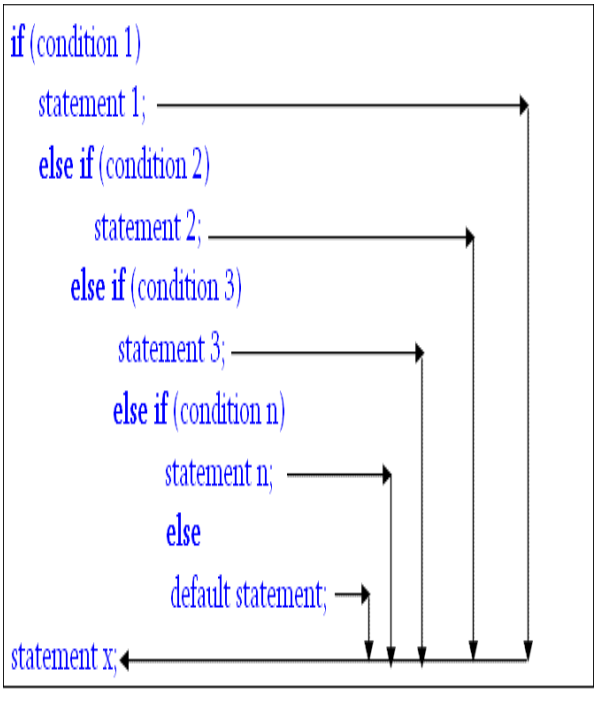
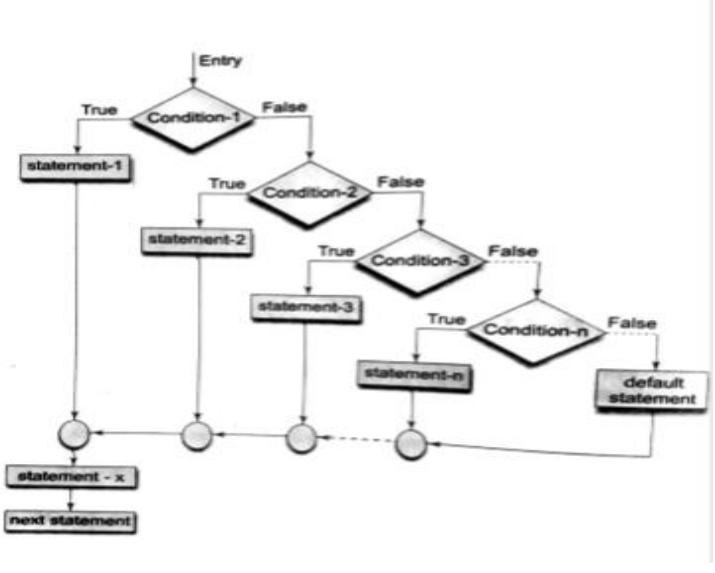


Fig:Flowchart



Program (greatest of three numbers)

```
#include<stdio.h>
int main()
{
    float a, b, c;
```

```
        printf("Enter three values\n");
        scanf("%f %f %f", &a, &b, &b);
        if(a>b && a>c)
        printf("%f is greatest", a);
        else if(b>c)
        printf("%f is greatest", b);
        else
        printf("%f is greatest", c);
        return 0;
}
```

Program (zero, positive or negative)

```
#include<stdio.h>
int main()
{
int a;
printf("Enter a value");
scanf("%d", &a);
if(a==0)
printf("%d is zero", a);
else if(a>0)
printf("%d is positive", a);
else
printf("%d is negative", a);
return 0;
}
```

Program (vowel or consonant)

```
#include<stdio.h>
int main()
{
        char a;
        printf("Enter a character");
        scanf("%c", &a);
        a=tolower(a);
```

```
        if(a==’a’)
                printf("%c is a vowel", a);
        else if(a==’e’)
                printf("%c is a vowel", a);
        else if(a==’i’)
                printf("%c is a vowel", a);
        else if(a==’o’)
                printf("%c is a vowel", a);
        else if(a==’u’)
                printf("%c is a vowel", a);
        else
                printf("%c is not a vowel or it is a consonant", a);
        return 0;
  }
```

Program:

Consider the following as grading of academic institution: Find the grade of a student if marks is given as input.

Average marks	**Grade**
80-100	Honours
60-79	First division
50-59	Second division
40-49	Third division
0-39	Fail

```
#include<stdio.h>
int main()
{
        int marks;
        printf("Enter the marks of a student");
        scanf("%d", &marks);
        if(marks>=80)
                printf(“honours");
        else if(marks>=60)
                printf(“First division");
        else if(marks>=50)
```

```
            printf("Second division");
        else if(marks>=40)
            printf(Third division");
        else
            printf("Fail");
        return 0;
}
```

Program - Compute the roots of a quadratic equation by accepting the coefficients. Print appropriate messages.

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
  int  main()
  {
        float a,b,c, d,root1,root2,real,imag;
        printf("Enter the three coefficients:\n");
        scanf("%f%f%f",&a,&b,&c);
        if(a==0)
        {
            printf("Invalid coefficients");
            exit(0);
        }
        d=b*b-4*a*c;
        if(d>0)
        {
            root1=(-b+(sqrt(d)))/(2.0*a);
            root2=(-b-(sqrt(d)))/(2.0*a);
            printf("The roots are real and distinct....\n");
            printf("root1=%f \n root2=%f\n",root1,root2);
        }
        else if(d==0)
        {
            root1=root2=-b/(2.0*a);
            printf("The roots are real and equal....\n");
            printf("root1=%f \n root2=%f\n",root1,root2);
        }
        else
        {
            real=-b/(2.0*a);
            imag= sqrt(fabs(d))/(2.0*a);
            printf("The roots are complex and imaginary....\n");
            printf("root1=%f+i %f \n root2= %f-i%f",real,imag,real,imag);
        }
        return 0;
  }
```

Program - An electricity board charges the following rates for the use of electricity: for the first 200

units 80 paise per unit: for the next 100 units 90 paise per unit: beyond 300 units Rs 1 per unit. All users are charged a minimum of Rs. 100 as meter charge. If the total amount is more than Rs 400, then an additional surcharge of 15% of total amount is charged. Write a program to read the name of the user, number of units consumed and print out the charges.

```
#include<stdio.h>
int main()
{
        char name[20];
        int units;
        float charges=0;
        printf("\n enter the name of the user :");
        gets(name);
        printf("\n enter number of units consumed :");
        scanf("%d",&units);
        if(units<=200)
        {
                charges=units*0.80;
        }
        else if(units<=300 && units>200)
        {
                charges=200*0.80+(units-200)*0.90;
        }
        else
        {
                charges=200*0.80+100*0.90+(units-300)*1.00;
        }
        charges=charges+100;
        if(charges>400)
                charges=charges+0.15*charges;
        printf("%s has to pay rupees %f",name,charges);
        return 0;
}
```

THE SWITCH STATEMENT

C has a built-in multi-way decision statement known as a **switch.** The **switch** statement tests the value of a given variable (or expression) against a list of **case** values and when a match is found, a block of statement associated with that **case** is executed.

The general form of the switch statement is as shown below

```
switch (expression)
{
        case value-1:
                Statement Block-1;
                break;
        case value-2:
                Statement Block-2;
                break;
……..
……
        default:
           default-block;
           break;
}
Statement-x;
```

The expression evaluates to an integral value (ie) an integer or character value. Value-1, Value-2….are constants known as case labels. Each of these values should be unique within a **switch** statement. Statement block-1, block-2…are statement list and may contain zero or more statements. There is no need to put braces around these blocks. **case** has labels end with a colon **(:)**

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.

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. **break** keyword indicates case is ended and control moves out of switch.

The selection process of **switch** statement is illustrated in the flow chart shown

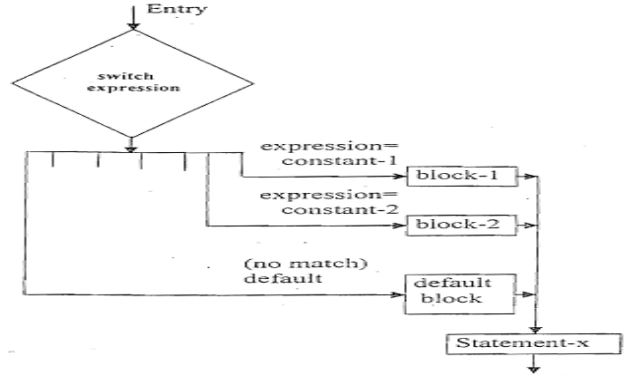


Fig: Flowchart

Advantages of Using a* switch case *Statement

Switch case statement is preferred by programmers due to the following reasons:

- Easy to debug
- Easy to read and understand
- Ease of maintenance as compared with its equivalent if-else statements
- Like if-else statements, switch statements can also be nested
- Executes faster than its equivalent if-else construct

Program:

Consider the following as grading of academic institution: Find the grade of a student if marks is given as input.

Average marks	**Grade**
80-100	Honours
60-79	First division
50-59	Second division
40-49	Third division
0-39	Fail

The switch statement can be used to grade the students.

```
#include<stdio.h>
int main()
{
    int marks,index;
    printf("Enter the marks of a student");
    scanf("%d", &marks);
    index=marks/10;
    switch (index)
    {
        case 10:
        case 9:
        case 8:
            printf( "\n Honours");
            break;
        case 7:
        Case 6:
```

```
                printf( "\n First Division");
                break;
        case 5:
                printf( "\n Second Division");
                break;
        case 4:
                printf( "\n Third Division");
                break;
        default:
                printf( "\n Fail");
    }
    return 0;
}
```

Program: Simulation of a Simple Calculator.

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
        int a,b,res;
        char op;
        printf("\n Enter a simple arithmetic expression");
        scanf("%d%c%d",&a,&op,&b);
        switch(op)
        {
                case '+':
                        res=a+b;
                        break;
                case '-':
                        res=a-b;
                        break;
                case '*':
                        res=a*b;
                        break;
                case '/':
```

```
                    if(b!=0)
                    res=a/b;
                    else
                    {
                        printf("division by zero is not possible");
                        exit(0);
                    }
                    break;
                case '%':
                    res=a%b;
                    break;
                default:
                    printf("illegal operator");
                    exit(0);
            }
            printf("\n%d%c%d=%d",a,op,b,res);
            return 0;
        }
```

Program (vowel or consonant)

```
#include<stdio.h>
#include<ctype.h>
int main()
  {
      char a;
      printf("Enter a character");
      scanf("%c", &a);
      a=tolower(a);
      switch(a)
      {
          case ‘a’:
          case ‘e’:
          case ‘i’:
          case ‘o’:
          case ‘u’:
```

```
                printf("%c is a vowel", a);
                break;
        default:
                printf("%c is not a vowel or consonant", a);
    }
    return 0;
}
```

THE ? : OPERATOR (The Ternary Operator)

The C language has an unusual operator, useful for making two-way decisions. This 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 condition expression. Otherwise, expression2 is evaluated and its value is returned. For example, the segment

```
if(a>b)
    big=a;
else
    big=b;
```

Can be written as

```
    big=(a>b) ? a : b;
```

Conditional operator can be nested for evaluating complex conditions.

Advantages: When the conditional operator is used, the code becomes more concise and perhaps, more efficient.

Disadvantages: readability is poor.

Program: Smallest of two numbers

```
#include<stdio.h>
int main()
{
    int a,b,small;
    printf("enter two numbers");
    scanf("%d%d",&a,&b);
    small=(a<b)?a:b;
```

```
        printf("%d is smallest",small);
        return 0;
}
```

Program: Smallest of three numbers

```
#include<stdio.h>
int main()
{
        int a,b,c,small;
        printf("enter three numbers");
        scanf("%d%d%d",&a,&b,&c);
        small=(a<b&&a<c)?a:(b<c?b:c);
        printf("%d is smallest",small);
        return 0;
}
```

Decision making and Looping (Iterative statements) – executes one or more statements repeatedly until some condition is met. **(while, do while, for)**

A program loop therefore consists of two segments,

1. body of the loop – statements to be repeatedly executed.
2. control statement – specifies a test condition to perform the body of loop.

Depending on the position of the control statement in the loop, a control structure may be classified either as the **entry-controlled loop or as the exit-controlled loop.**

In the entry controlled loop, the control conditions are tested before the start of the loop execution. If the conditions are not satisfied, and then the body of the loop will not be executed.

In the case of an exit-controlled loop, the test is performed at the end of the body of the loop and therefore the body is executed unconditionally for the first time. The entry-controlled loops and exit controlled loops also known as **pre-test and post-test loops respectively.**

Entry-controlled loop	**Exit- controlled loop**
Control conditions are tested before the body of the loop	Control conditions are tested after the body of the loop
If the control condition is false for the first time, body of the loop is never executed	Body of the loop is executed at least one time irrespective of the condition
It is called pre-test loop	It is called post-test loop
Example – while, for	Example – do while

Flow-chart shown below	Flow-chart shown below

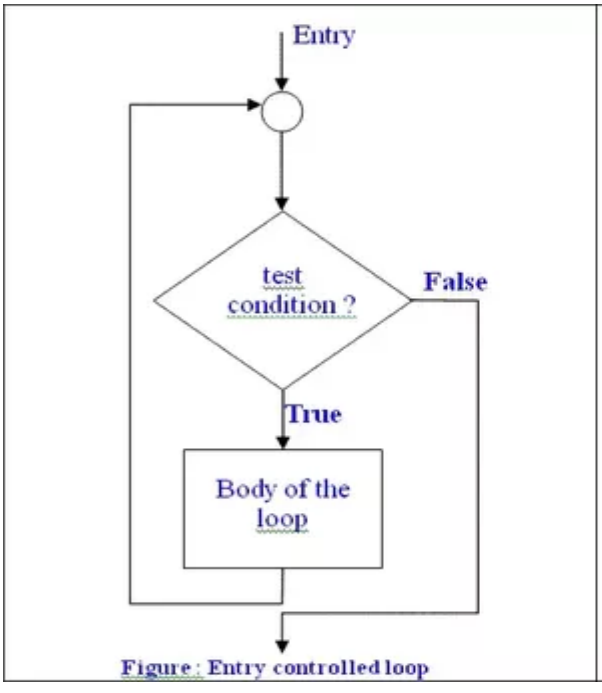


Figure : Entry controlled loop

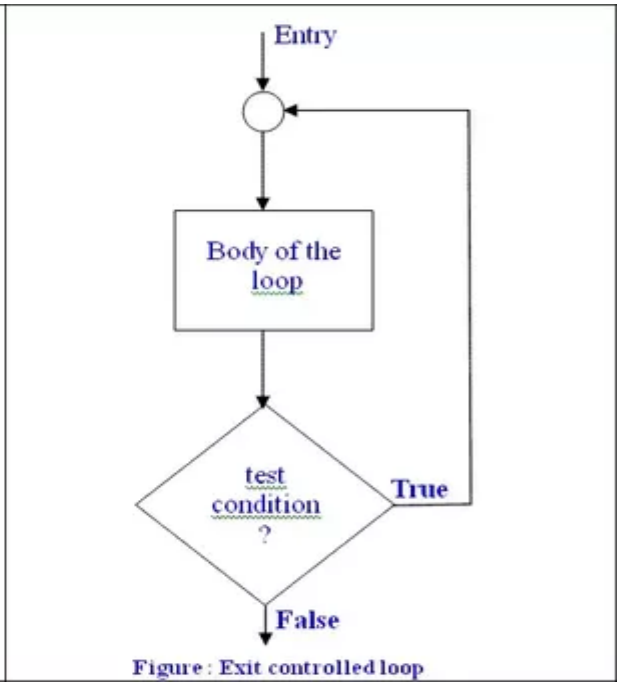


Figure : Exit controlled loop

Loops enter to infinite loop, if the condition does not fail.

A looping process would include the following four steps:

1. Setting and initialization of a condition variable or control variable.
2. Execution of statements in the loop (body of the loop).
3. Test for a specified value of the control variable for execution of the loop.
4. Incrementing or updating the condition variable.

The C language provides for three constructs for performing loop operations

1. The **while** statement
2. The **do while** statement
3. The **for** statement

THE WHILE STATEMENT

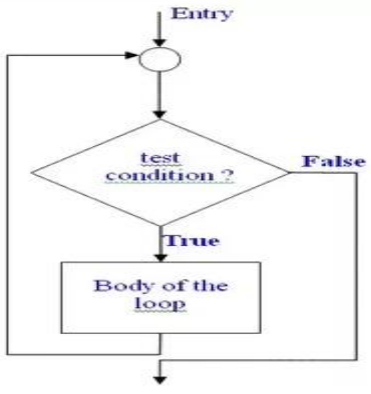
The simplest of all the looping structures in C is the while statement. The while is an entry-controlled loop statement.

Syntax:

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

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. On exit, the program continues with the statement immediately after the body of the loop.



Program (print first n natural numbers)

```
#include<stdio.h>
int main()
{
        int i,n;
        printf("enter n:");
        scanf("%d",&n);
        i=1;
        while(i<=n)
        {
                printf("%d\t",i);
                i++;
        }
        return 0;
}
```

Program (print first n natural numbers in reverse)

```
#include<stdio.h>
int main()
{
        int i,n;
        printf("enter n:");
        scanf("%d",&n);
```

```
        i=n;
        while(i>=1)
        {
                printf("%d\t",i);
                i--;
        }
        return 0;
}
```

Program (print the sum of first n numbers)

```
#include<stdio.h>
int main()
{
        int i,n,sum=0;
        printf("enter n:");
        scanf("%d",&n);
        i=1;
        while(i<=n)
        {
                sum=sum+i;
                i=i+1;
        }
        printf("sum=%d",sum);
        return 0;
}
```

Program (print the sum of first n squares)

```
#include<stdio.h>
void main()
{
        int i,n,sum=0;
        printf("enter n:");
        scanf("%d",&n);
        i=1;
        while(i<=n)
        {
```

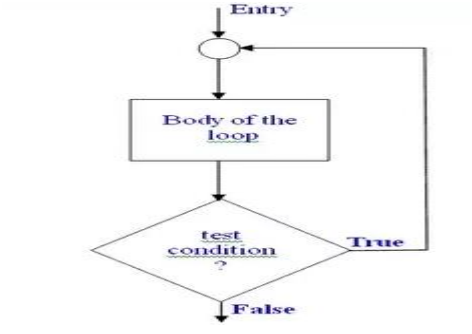
```
        sum=sum+i*i;
        i=i+1;
    }
    printf("sum=%d",sum);
}
```

THE do while LOOPING STATEMENT

On some, occasions it might be necessary to execute to 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);
```



Since the test-condition is evaluated at the bottom of the loop, the **do…while** is an exit-controlled loop.

Program (print first n natural numbers)

```
#include<stdio.h>
int main()
{
    int i,n;
    printf("enter n:");
    scanf("%d",&n);
    i=1;
    do
    {
        printf("%d\t",i);
        i++;
```

```
    }
    while(i<=n);
    return 0;
}
```

THE FOR STATEMENT

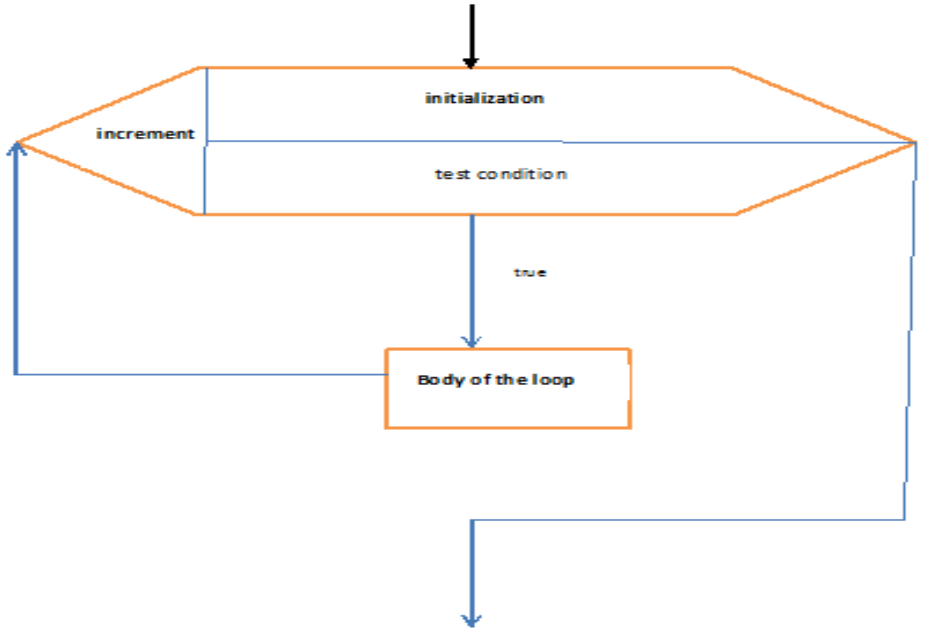
Simple 'for' Loops

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

The general form of the **for** loop is

Syntax:

```
for(initialization; test-condition ; increment)
{
        body of the loop;
}
```



The execution of the for statement is as follows

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 at 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 Incremented using an assignment statement such as i=i+1 and the new value of the control variable is again tested to see whether it satisfies

the loop condition. If the condition is satisfied, the body of the loop is again executed. This process continues till the value of the control variable fails to satisfy the test-condition.

Program (print first n natural numbers)

```
#include<stdio.h>
int main()
{
        int i,n;
        printf("enter n:");
        scanf("%d",&n);
        for(i=1;i<=n;i++)
        {
                printf("%d\t",i);
        }
        return 0;
}
```

Program (print even numbers from 2 till n)

```
#include<stdio.h>
int main()
{
        int i,n;
        printf("enter n:");
        scanf("%d",&n);
        for(i=2;i<=n;i=i+2)
        {
                printf("%d\t",i);
        }
        return 0;
}
```

Program (print the sum of first n numbers)

```
#include<stdio.h>
int main()
{
        int i,n,sum=0;
        printf("enter n:");
```

```
        scanf("%d",&n);
        for(i=1;i<=n;i++)
        {
                sum=sum+i;
        }
        printf("sum=%d",sum);
        return 0;
}
```

Program (print the factorial of n)

```
#include<stdio.h>
int main()
{
        int i,n,f=1;
        printf("enter n:");
        scanf("%d",&n);
        for(i=1;i<=n;i++)
        {
                f=f*i;
        }
        printf("factorial of %d =%d",n,f);
        return 0;
}
```

Additional Features of for loop

1. More than one variable can be initialized at a time in the **for** statement.

```
for (p=1,n=0; n<17; ++n)
```

2. Like the initialization section, the increment section may also have more than one part. For example the loop

```
for(n=1, m=50; n<=m; n=n+1, m=m-1)
{
        p=m/n;
        printf("%d %d %d\n", n, m, p);
}
```

   is perfectly valid. The multiple arguments in the Increment section are separated by commas.

3. The third feature is that the test-condition may have any compound relation and the testing need not

be limited only to the loop control variable. Consider the example below:

```
sum 0;
for (i =1; i < 20 && sum < 100;  ++i)
{
        sum = sum+i;
        printf("%d %d\n", i, sum);
}
```

4. Another unique aspect of for loop is that one or more sections can be omitted, if necessary. Consider the following statements

```
m=5;
for (; m !=100;)
{
        printf("%d\n",m);
        m=m+5;
}
```

5. we can set up time delay using the null statement(;) as follows:

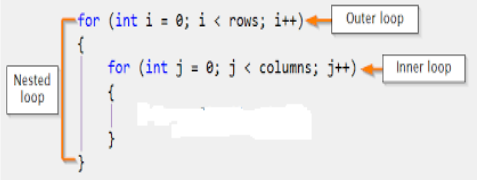
```
for(i=1;i<=1000;i++);
```

The loop is executed 1000 times without producing any output; it simply causes a time delay.

Nesting of for loop

Nesting of for loops that is one **for** statement within another **for** statement**,** is allowed in C. For example two loops can be nested as follows:

Syntax:



Write a C program using nested for loop to print the following pattern.

i)
```
*
* *
* * *
* * * *
```

ii)
```
* * * *
* * *
* *
*
```

```
#include<stdio.h>
int main()
{
```

```
                int  i,j;
                for (i=1;i<=4; i++)
                {
                        for (j=1;j<=i; j++)
                        {
                                printf("*\t");
                        }
                        printf("\n");
                }
                return 0;
        }
        #include<stdio.h>
        int main()
        {
                int  i,j;
                for (i=4;i>=1; i--)
                {
                        for (j=1;j<=i; j++)
                        {
                                printf("*\t");
                        }
                        printf("\n");
                }
                return 0;
        }
```

Just like for loop, even while and do while can be nested.

```
#include<stdio.h>
int main()
{
        int x,y,sum;
        x=1;
        while(x<=2)
        {
                y=1;
```

```
            while(y<=2)
            {
                  sum=x+y;
                  printf("\nx=%d",x);
                  printf("\ny=%d",y);
                  printf("\nsum=%d",sum);
                  y++;
            }
            x++;
      }
}
```

Output:

x=1
y=1
sum=2
x=1
y=2
sum=3
x=2
y=1
sum=3
x=2
y=2
sum=4

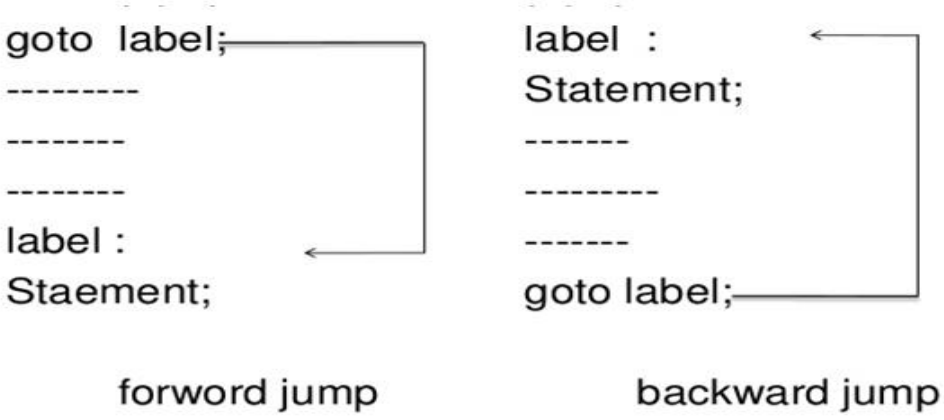
Decision Making and Unconditional jump statements in C:

goto, break, continue, exit, return

THE GOTO STATEMENT

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:



During running of a program when a statement like - **goto begin;** is met, the flow of control will jump to the statement immediately following the label begin: This happens unconditionally.

If the label: is before the statement **goto** label; a loop will be formed and some statements will be executed repeatedly. Such a jump is known as a backward jump. On the other hand, if the label: is placed after the goto label; some statements will be skipped and the jump is known as forward jump.

Program (use goto to write a program to perform the sum of first n natural numbers) - Backward jump works like loop.

```
#include<stdio.h>
int main()
{
        int i,n,sum;
        printf("enter n");
        scanf("%d",&n);
        sum=0;
        i=0;
        loop:
        sum=sum+i;
        i=i+1;
        if(i<=n)
                goto loop;
        printf("sum=%d", sum);
        return 0;
}
```

Program (use goto to write a program to check if the input marks is pass or fail)

```
#include<stdio.h>
int main()
```

```
{
        int marks;
        printf("enter marks");
        scanf("%d",&marks);
        if(marks<50)
                goto fail;
        else
                goto pass;
        fail:
                printf("failed");
                exit(0);
        pass:
                printf("passed");
        return 0;
}
```

Avoiding goto

It is a good practice to avoid using **goto.** Using many of them makes a program logic complicated and renders the program unreadable. The **goto** jumps shown in below fig would cause problems and therefore must be avoided

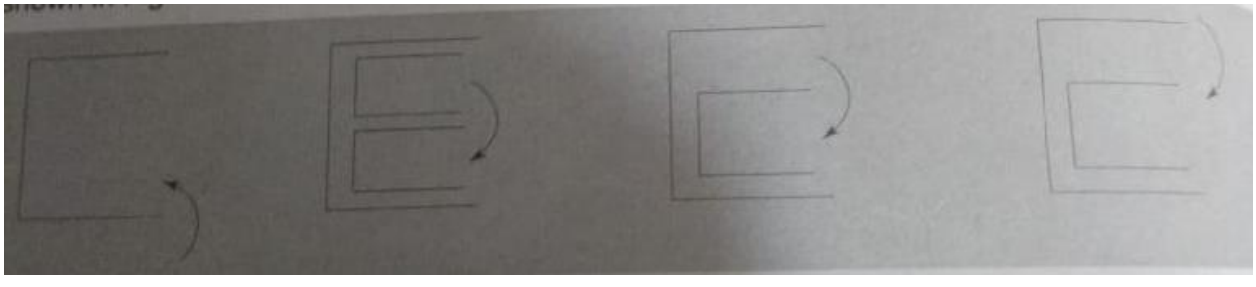


Fig: Avoid goto

Jumping out of a loop using break statement

An early exit from a loop can be accomplished by using the **break** statement or the **goto** statement.

Syntax for break: break;

These statements can also be used within **while, do,** or **for** loops. When a **break** statement-is encountered inside a loop, the loop is immediately exited and the program continues with the statement immediately following the loop.

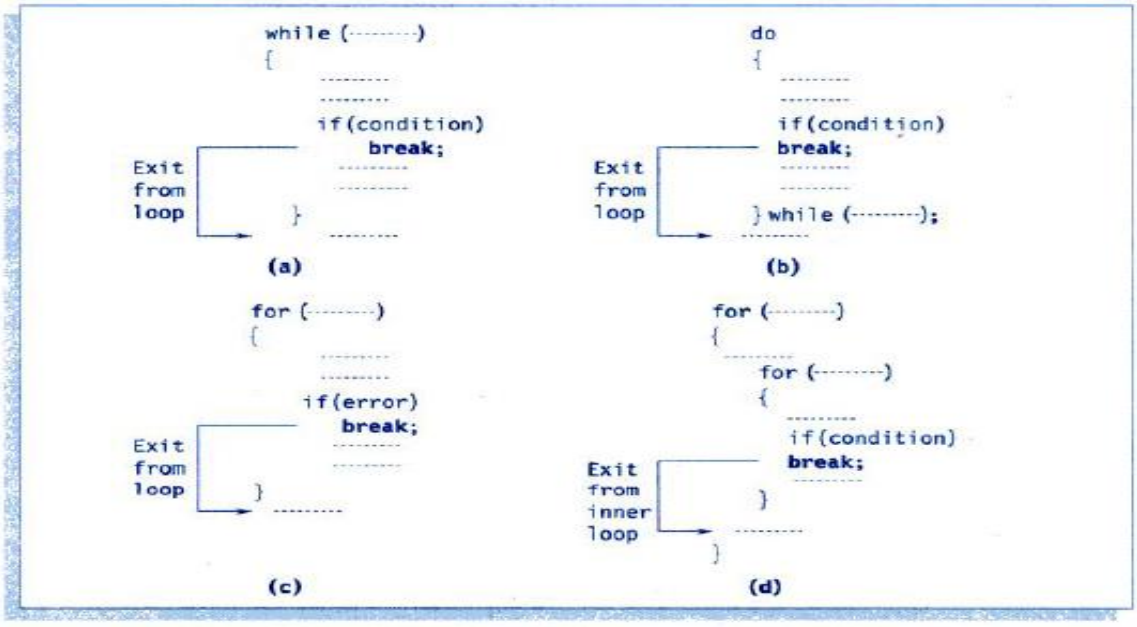


Fig. 6.6 *Exiting a loop with* ***break*** *statement*

Program (to print first 10 even numbers not greater than 10)

```
#include<stdio.h>
int main()
{
        int i;
        for(i=2;i<=20;i=i+2)
        {
                if(i>10)
                break;
                printf("%d\t",i);
        }
        return 0;
}
```

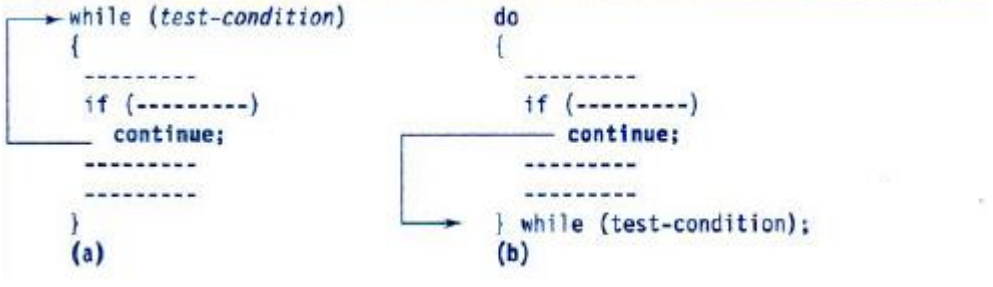
Skipping a part of loop using continue statement

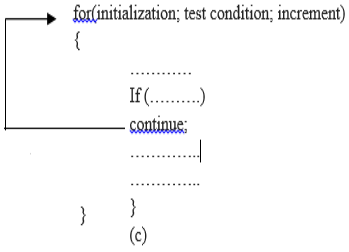
During the loop operations, it may be necessary to skip a part of the body of the loop under certain conditions. Unlike the **break** which causes the loop to be terminated, the continue, causes the loop to be continued with the next iteration after skipping any statements in between. The **continue** statement tells the compiler, "SKIP THE FOLLOWING STATEMENTS AND CONTINUE WITH THE NEXT ITERATION". The format of the continue statement is simply

Syntax:

```
continue;
```

The use of the **continue** statement in loops is Illustrated





Program (to print first 10 even numbers except 6 and 8)

```
#include<stdio.h>
int main()
{
        int i;
        for(i=2;i<=20;i=i+2)
        {
                if(i==6||i==8)
                continue;
                printf("%d\t",i);
        }
        return 0;
}
```

Jumping out of the program – exit()

We have just seen that we can jump out of a loop using either the **break** statement or **goto** statement. In a similar way, we can jump out of a program by using the library function **exit( ).** In case, due to some reason, we wish to break out of a program and return to the operating system, we can use the exit() function, as shown below

```
……
……...
if (test-condition)
        exit(0);
……..
……..
```

The **exit()** function takes an integer value as its argument. Normally zero is used to indicate normal termination and a nonzero value to indicate termination due to some error or abnormal condition. The use of **exit()** function requires the inclusion of the header file **<stdlib.h>**

Return statement: return statement terminates the execution and returns control to the calling function.

Syntax: return expression;

Example: return 0; 0 will be returned to calling function.

Chapter 1: Operators in C, Type Conversion and Typecasting

An **expression** is a sequence of operands and operators that reduces to a single value.

An **operation** is specified as an expression that reduces to a single value.

An **operand** is the data item on which an operation is done.

An **operator** is a symbol that tells computer to perform certain mathematical or logical operations.

An operation can be **unary, binary or ternary**. Unary consists of single operand, Binary consists of two operands and ternary consists of three operands. All operations have at least one operator.

C operators can be classified into number of categories. They include:

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

1 ARITHMETIC OPERATORS

C provides all the basic arithmetic operators. The operators +,-,*, and / all work the same way as they do in the other languages. The unary minus operator, in effect, multiples its single operand by -1. Therefore, a number preceded by a minus sign changes its sign.

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

Integer division truncates any fractional part. The modulo division operation produces the remainder of an integer division. Examples of use of arithmetic operators are:

a-b	a+b
a*b	a/b
a%b	-a*b

Here **a** and **b** are variables and are known as *operands*. The modulo division operator % cannot be used on floating point data.

Integer Arithmetic

When both the operands in a single arithmetic expression such as a+b are integers, the expression is called an *integer expression*, and the operation is called *integer arithmetic*. Integer arithmetic always yields an integer value. The largest integer values depends on the machine, as pointed out earlier. In the above examples, if **a** and **b**are integers, then for **a**=14 and **b**=4 we have the following results:

a – b = 10

a + b = 18

a * b = 56

a / b = 3 (decimal part truncated)

a % b = 2 (remainder of division)

Similarly, during modulo division, the sign of the result is always the sign of the first operand (the dividend). That is,

-14 % 3 = -2

-14 % -3 = -2

14 % -3 = 2

Real Arithmetic / Floating Point arithmetic

An arithmetic operation involving only real operands is called *real arithmetic*. If **x**, **y**, and **z** are **floats**, then we will have:

x = 6.0 / 7.0 = 0.857143

y = 1.0 / 3.0 = 0.333333

z = -2.0 / 3.0 = -0.666667

The operator % cannot be used with real operands.

Mixed-mode Arithmetic

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

15 / 10.0 = 1.5

Whereas

15 / 10 = 1

Program: Addition of two numbers

```
#include<stdio.h>
int main()
{
    int a=10,b=20,c;
    c=a+b;
    printf("result=%d",c);
    return 0;
}
```

2 RELATIONAL OPERATORS

We often compare two quantities depending on their relation, take certain decisions. These comparisons can be done with the help of *relational operators*. We have already used the symbol '<' , meaning 'less than'. An expression such as

a<b or 1<20

Containing a relational operator is termed as a *relational expression*. The value of a relational expression is either *one* or *zero*. It is *one* if the specified relation is *true* and *zero* if the relation is *false.*

For example

10 < 20 is true

but

20 < 10 is false

C supports six relational operators in all. These operators and their meanings are shown under.

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

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

ae-1 relational operator ae-2

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

4.5 <= 10 TRUE

4.5 <-10 FALSE

When arithmetic expressions are used on either side of a relational operator, the arithmetic expressions will be evaluated first and then the results compared. That is, 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

Program: Greatest of two numbers

```
#include<stdio.h>
int main()
{
        int a,b;
        printf("enter two numbers");
        scanf("%d%d",&a,&b);
        if(a>b)
                printf("%d is greatest",a);
        else
                printf("%d is greatest",b);
        return 0;
}
```

3 LOGICAL OPERATORS

In addition to the relational operators, C has the following three *logical operators*.

&& meaning logical AND

| | meaning logical OR

! meaning logical NOT or Logical negation

The logical operators && and | | are used when we want to test more than one condition and make decisions. An example is:

a > b && x==10

An expression of this kind, which combines two or more relational expressions, is termed as a *logical expression.*

op-1	op-2	Value of the expression	
		op-1 && op-2	op-1 || op-2
Non-zero	Non-zero	1	1
Non-zero	0	0	1
0	Non-zero	0	1
0	0	0	0

op1	!op1
0	1
1	0

Program: Greatest of three numbers

```
#include<stdio.h>
int main()
{
int a,b,c;
```

```
printf("enter three numbers");
scanf("%d%d%d",&a,&b,&c);
if((a>b)&&(a>c))
        printf("%d is greatest",a);
else if(b>c)
        printf("%d is greatest",b);
else
        printf("%d is greatest",c);
return 0;
}
```

4 ASSIGNMENT OPERATORS

Assignment operators are used to assign the result of an expression to a variable. We have seen the usual assignment operator, '='.

v =exp;

In addition, C has a set of '*shorthand*' assignment operators of the form:

v op= exp;

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

The assignment statement

v op= exp;

is equivalent to

v = v op (exp);

Consider an example **x += y;**

This is the same as the statement **x = x + y;**

Operators	Example/Description
=	sum = 10; 10 is assigned to variable sum
+=	a += 10; This is same as a = a

	+ 10
-=	a - = 10; This is same as a = a – 10
*=	a *= 10; This is same as a = a * 10
/=	a /= 10; This is same as a = a / 10
%=	a %= 10; This is same as a = a % 10
&=	a &= 10; This is same as a = a & 10
|=	a |= 10; This is same as a = a | 10
^	a ^= 10; This is same as a = a ^ 10
<<	a <<= 10; This is same as a = a << 10
>>	a >>= 10; This is same as a = a >> 10

The use of shorthand assignment operators has three 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: Addition of two numbers

```
int main()
{
int a=10,b=20;
a+=b;
printf("result=%d",a);
return 0;
}
```

5 INCREMENT AND DECREMENT OPERATORS

The increment and decrement operators:

++ and– –

The operator ++ adds 1 to the operand, while – – subtracts 1. Both are unary operators and takes the following form:

++a; or a++;

– –a; or a– –;

++a; is equivalent to a = a+1; (or a +=1;)

– –a; is equivalent to a = a-1; (or a – =1;)

A prefix operator first adds 1 to the operand and then the result is assigned to the variable on the left. On the other hand, a postfix operator first assigns the value to the variable on left and then increments the operand.

y=++x	*x=x+1* *y=x* *Ex : x=10* *x=11* *y=11*	*y=x++*	*y=x* *x=x+1* *Ex : x=10* *y=10* *x=11*
y=--x	*x=x-1* *y=x* *Ex : x=10* *x=9*	*y=x--*	*y=x* *x=x-1* *Ex : x=10* *y=10*

	y=9		*x=9*

Program:

```
#include<stdio.h>
int main()
{
    int a=10;
    printf("%d",a++);
    return 0;
}
```

Output: 10

6 CONDITIONAL OPERATOR

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

exp1 ?exp2 : exp3

where exp1, exp2 and exp3 are expressions.

The operator ? : works as follows: *exp1* is evaluated first, if it is nonzero (true), then the expression *exp2* is evaluated and becomes the value of the expression. If *exp1* is false, *exp3* is evaluated and its value becomes the value of the expression.

```
a = 10;
b = 15;
big = (a>b) ?a : b;
```

In this example, big will be assigned the value of b. This can be achieved using the **if..else** statements as follows:

```
if (a > b)
        big = a;
else
        big = b;
```

Program: Smallest of two numbers

```
int main()
{
inta,b,small;
printf("enter two numbers");
scanf("%d%d",&a,&b);
```

```
small=a<b?a:b;
printf("%d is smallest",small);
return 0;
}
```

Program: Smallest of three numbers

```
int main()
{
inta,b,c,small;
printf("enter three numbers");
scanf("%d%d%d",&a,&b,&c);
small=a<b&&a<c?a:(b<c?b:c);
printf("%d is smallest",small);
return 0;
}
```

7 BITWISE OPERATORS

C has 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 operators may not be applied to **float** or **double**. **(bitwise complement ~ is also a bitwise operator which complements the bit (ie) complement of 0 is 1, complement of 1 is 0.)**

Operator	*Meaning*
&	bitwise AND
|	bitwise OR
^	bitwise exclusive OR
<<	shift left
>>	shift right

op1	**op2**	**op1&op2**	**op1|op2**	**op1^op2**
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

op1	~op1
0	1
1	0

8&2=0 (i.e 1000&0010=0000)

8|2=10 (i.e 1000|0010=1010)

8^2=10 (i.e 1000^0010=1010)

8<<2=32 (shift bit by bit)

1000

10000(by one bit to left)

100000(by next one bit to left)=32

8>>2=2 (shift bit by bit)

1000

0100(by one bit to right)

0010(by next one bit to right)=2

Program:

```
int main()
{
        int a=8,b=2;
        printf("%d", a&b);
        return 0;
}
```

8 SPECIAL OPERATORS

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

The Comma Operator

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

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

first assigns the value 10 to **x**, then assigns 5 to **y,** and finally assigns 15 (i.e. 10 + 5) to **value**. Since comma operator has the lowest precedence of all operators, the parentheses are necessary. Some applications of comma operator are:

The sizeof Operator

The **sizeof** returns the number of bytes the operand occupies. The operand may be a variable, a constant or a data type qualifier.

Examples:

m = **sizeof**(int); 2 bytes in a 16-bit machine

n =**sizeof**(long int); 4 bytes in a 16-bit machine

k=sizeof(100); 2 bytes in a 16-bit machine

Program:

```
#include<stdio.h>
int main()
{
        printf("size of integer is %d",sizeof(int));
        return 0;
}
```

Mathematical functions

Mathematical functions such as cos, sqrt, log, etc. are frequently used in analysis of real-life problems. Most of the C compiles support these basic math functions. To use these functions in a program, we should include the line:

#include <math.h> in the beginning of the program.

Some of the Math functions are

- cos(x) cosine of x
- sin(x) sine of x
- tan(x) tangent of x
- cosh(x) hyberbolic cosine of x
- sinh(x) hyperbolic sine of x
- tanh(x) hyperbolic tangent of x
- ceil(x) x rounded to nearest integer
- exp(x) e to the x power (e^x)
- abs(x) absolute value of x
- fabs(x) absolute value of float x
- log(x) natural log of x, x>0
- pow(x,y) x to the power y (x^y)
- sqrt(x) square root of x, x>=0

ARITHMETIC EXPRESSIONS

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

Table 3.6 *Expressions*

Algebraic expression	*C expression*
a x 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

OPERATOR PRECEDENCE AND ASSOCIATIVITY

C operators has a precedence associated with it. This precedence is used to determine how an expression involving more than one operator is evaluated. There are distinct *levels of precedence* and an operator may belong to one of these levels. The operators at the higher level of precedence are evaluated first. The operators of the same precedence are evaluated either from 'left to right' or from 'right to left', depending on the level. This is known as the *associativity* property of an operator. Rank 1 indicates the highest precedence level and 15 the lowest.

Rules of precedence and Associativity

- Precedence rules decides the order in which different operators are applied
- Associativity rule decides the order in which multiple occurrences of the same level operator are applied.

Operator	*Description*	*Associativity*	*Rank*
( ) [ ]	Function call Aray element reference	Left to right	1
+ − ++ −− ! ~ * & sizeof (type)	Unary plus Unary minus Increment Decrement Logical negation Ones complement Pointer referencé (indirection) Address Size of an object Type cast (conversion)	Right to left	2
* / %	Multiplication Division Modulus	Left to right	3
+ −	Addition Subtraction	Left to right	4
<< >>	Left shift Right shift	Left to right	5
< <= > >=	Less than Less than or equal to Greater than Greater than or equal to	Left to right	6
== |=	Equality Inequality	Left to right	7
&	Bitwise AND	Left to right	8
^	Bitwise XOR	Left to right	9
|	Bitwise OR	Left to right	10
&&	Logical AND	Left to right	11
||	Logical OR	Left to right	12
?:	Conditional expression	Right to left	13
= * = /= %= += −= &= ^= |= <<= >>=	Assignment operators	Right to left	14
,	Comma operator	Left to right	15

SOME COMPUTATIONAL PROBLEMS

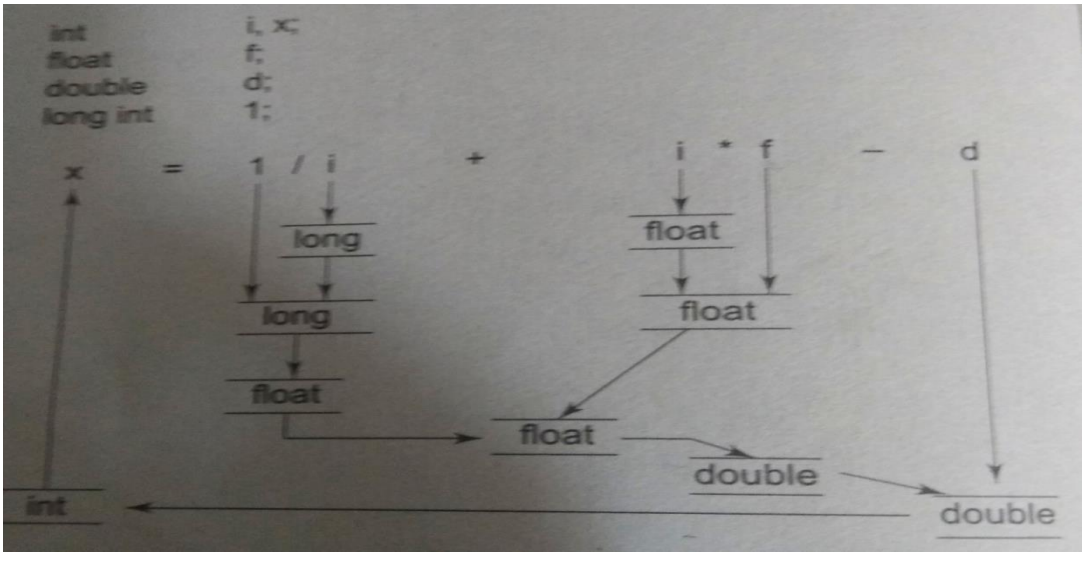
- When expressions include real values, then it is important to take necessary precautions to guard against certain computational errors due to approximations.
- Another problem is division by zero. On most computers, any attempt to divide a number by zero will result in abnormal termination of the program.
- The third problem is to avoid overflow or underflow errors. It is our responsibility to guarantee that the operands are of the correct type and range, and the result may not produce any overflow or underflow.

TYPE CONVERSIONS IN EXPRESSIONS

Implicit Type Conversion

C permits mixing of constants and variables of different types in an expression. C automatically converts any intermediate values to the proper type so that the expression can be evaluated without losing any significance. This automatic conversion is known as *implicit type conversion*.

During evaluation it adheres to very strict rules of type conversion. If the operands are of different types, the 'lower' type is automatically converted to the 'higher' type before the operation proceeds. The result is of the higher type. A typical type conversion process is illustrated in the below Fig.



Given below is the sequence of promotion rules that are applied while evaluating expressions.

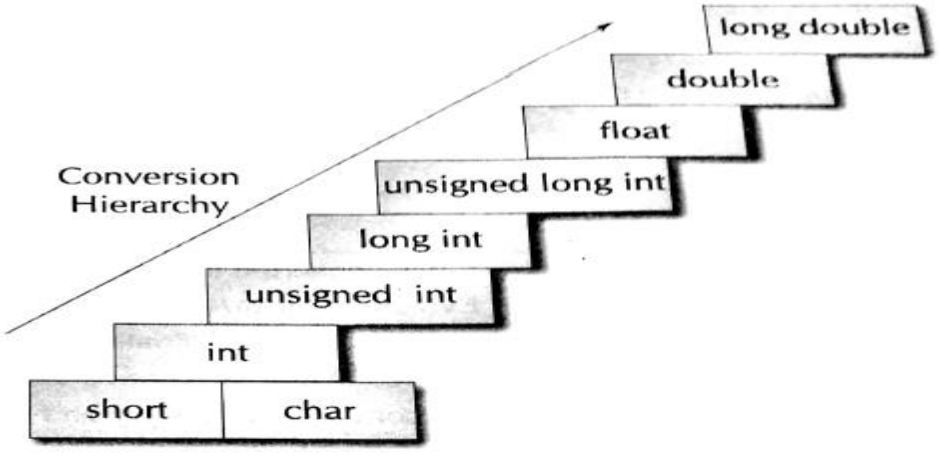
All **short** and **char** are automatically converted to **int**, then

1. If one of the operands is **long double**, the other will be converted to **long double** and the result will be **long double.**
2. Else, if one of the operands is **double**, the other will be converted to **double** and the result will be **double.**
3. Else, if one of the operands is **float**, the other will be converted to **float** and the result will be **float.**

4. Else, if one of the operands is **unsigned long int**, the other will be converted to **unsigned long int** and the result will be **unsigned long int;**
5. Else, if one of the operands is **long int** and the other is **unsigned int**, then
   (a) If **unsigned int** can be converted to **long int**, the **unsigned int** operand will be converted to **long int.**
   (b) Else, both operands will be converted to **unsigned long int** and the result will be **unsigned long int.**
6. Else, if one of the operands is **long int**, the other will be converted to **long int** and the result will be **long int.**
7. Else, if one of the operands is **unsigned int**, the other will be converted to **unsigned int** and the result will be **unsigned int.**

Conversion Hierarchy

Note that, C uses the rule that, in all expressions except assignments, any implicit type conversions are made from a lower size type to a higher size type as shown below:



The final result of an expression is converted to the type of the variable on the left of the assignment sign before assigning the value to it.

However, the following changes are introduced during the final assignment.

1. **float, double** to **int** causes truncation of the fractional part.
2. **double** to **float** causes rounding of digits.
3. **long int** to **int** causes dropping of the excess higher order bits.

```
int main()
{
int a=1;
```

```
float b=4,c;
c=a/b;
printf("%f",c);
return 0;
}
```

output : 0.25

Explicit Conversion

There are instances when we want to force a type conversion in a way that is different from the automatic conversion. Consider, for example, the calculation of ratio of females to males in a town.

ratio = female_number / male_number

Since **female_number** and **male_number** are declared as integers in the program, the decimal part of the result of the division would be lost and **ratio** would represent a wrong figure. This problem can be solved by converting locally one of the variables to the floating point as shown below:

ratio = **(float)** female_number / male_number

The operator **(float)** converts the **female_number** to floating point for the purpose of evaluation of the expression. Then using the rule of automatic conversion, the division is performed in floating point mode, thus retaining the fractional part of the result.

The process of such a local conversion is known as *explicit conversion* or *casting a value*. The general form of a cast is:

(type-name) expression

Where *type-name* is one of the standard C data types. The expression may be a constant, variable or an expression. Example

x= (int) 7.5 ; here 7.5 is converted to integer by truncation, (i.e) 7.

```
#include<stdio.h>
int main()
{
        int a=1,b=4;
        float c;
        c=(float)a/b;
        printf("%f",c);
        return 0;
}
```

output : **0.25**

#include<stdio.h> int main()	#include<stdio.h> int main()

{
int a=1,b=4,c;
c=a/b;
printf("%d",c);
}
 	
output: 0
	{
	int a=1;
	float b=4,c;
	c=a/b;
	printf("%f",c);
	}
	output: 0.25

#include<stdio.h>
int main()
{
int a=1,c;
float b=4;
c=a/b;
printf("%d",c);
return 0;
}
output: 0
	#include<stdio.h>
	int main()
	{
	float a=1,b=4,c;
	c=a/b;
	printf("%f",c);
	return 0;
	}
	output: 0.25

#include<stdio.h>
int main()
{
int a=1,b=4;
float c;
c=(float)a/b;
printf("%f",c);
return 0;
}
output: 0.25
	#include<stdio.h>
	int main()
	{
	int a=1,b=4;
	float c;
	c=a/(float)b;
	printf("%f",c);
	return 0;
	}
	output: 0.25

#include<stdio.h>
int main()
{
int a=1,b=4,c;
c=(float)a/b;
printf("%d",c);
return 0;
	#include<stdio.h>
	int main()
	{
	float a=1,b=4;
	int c;
	c=a/b;
	printf("%d",c);

}	return 0;
	}
output: 0	**output: 0**

EVALUATION OF EXPRESSIONS

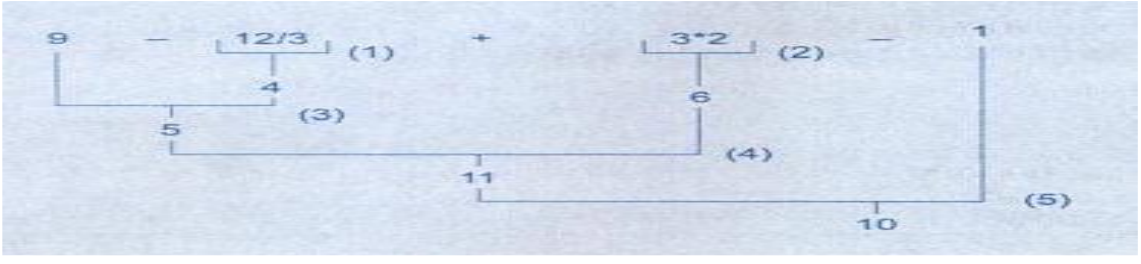
Expressions are evaluated using an assignment statement of the form:

Variable = expression;

Examples of evaluation statements are:

x = a * b – c;

The numbers inside parenthesis refer to step numbers.



Consider the same expression with parenthesis as shown below:

9 – 12 / (3 + 3) * (2 – 1)

Whenever parentheses are used, the expressions within parentheses assume highest priority.

Step 1: 9 – 12 / 6 * (2 – 1)

Step 2: 9 – 12 / 6 * 1

Step 3: 9 – 2 * 1

Step 4: 9 – 2

Step 5: 7

While parentheses allow us to change the order of priority, we may also use them to improve the understand-ability of the program.

- First, parenthesized sub expressions from left to right are evaluated.
- If parentheses are nested, the evaluation begins with the innermost sub-expression.
- The precedence rule is applied in determining the order of application of operators in evaluating sub-expressions.
- The associativity rule is applied when two or more operators of the same precedence level appear in a sub-expression.

Evaluate the following expressions:

(a) If a=3, d=7, e=2, c=5, b=4, x= --a*d/e-c++ * b, find a, b, c, d, e, x.

x= --a*d/e-c++ * b

x=--a*d/e-5*b (c++=5, c=c+1=5+1=6)

x=2*d/e-5*b (--a=2,a=2)

x=2*7/2-5*4

x=14/2-5*4 (2*7=14)

x=7-5*4 (14/2=7)

x=7-20 (5*4=20)

x=-13 (7-20=-13)

a=2

b=4

c=6

d=7

e=2

x=-13

(b) If x=4, y=3, z=2, m=++x + --y + z++ + --z, find m, x, y, z.

m=++x + --y + z++ + --z

m=++x + --y + z++ + 1(--z=1, z=1)

m=++x + --y + 1 + 1 (z++=1, z=z+1=2)

m=++x + 2 + 1 + 1 (--y=2, y=2)

m=5 + 2 + 1 + 1 (++x=5, x=5)

m=7+1+1 (5+2=7)

m=8+1 (7+1=8)

m=9 (8+1=9)

x=5

y=2

z=2

(c) if x=20, y=5, find the value of the expression x= =10+15&&y<10

x= =10+15&&y<10

20==25&&5<10 (10+15=25)

20==25&&1 (5<10 is true, which is 1)

0&&1(20==25 is false, which is 0)

0 (0&&1 is 0)

(d) if a=9,b=12,c=3, find the value of the expression a-b/3+c*2-1

a-b/3+c*2-1

9-12/3+3*2-1

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

9-4+6-1 (3*2=6)

5+6-1 (9-4=5)

11-1 (5+6=11)

10 (11-1=10)

(e) if a=9,b=12,c=3, find the value of the expression a-b/(3+c)*(2-1)

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

9-12/(3+3)*(2-1)

9-12/6*(2-1) [3+3=6]

9-12/6*1 [2-1=1]

9-2*1 [12/6=2]

9-2 [2*1=2]

7 [9-2=7]

(f) if a=9,b=12,c=3, find the value of the expression a-(b/(3+c)*2)-1

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

9-(12/(3+3)*2)-1

9-(12/6*2)-1 [3+3=6]

9-(2*2)-1 [12/6=2]

9-4-1 [2*2=4]

5-1 [9-4=5]

4 [5-1=4]

(g) if a=9,b=12,c=3, find the value of the expression a-((b/3)+c*2)-1

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

9-((12/3)+3*2)-1

9-(4+3*2)-1 [12/3=4]

9-(4+6)-1 [3*2=6]

9-10-1 [4+6=10]

-1-1 [9-10=-1]

-2 [-1-1=-2]

(h) 10!=10 || 5<4 && 8
10!=10 || 5<4 && 8
10!=10 || 0 && 8 (5<4 =0, false)
0||0 && 8 (10!=10 = 0, false)
0||0 (0 && 8 = 0, false)
0 (0 || 0 = 0, false)