Unit - 2

Chapter 7

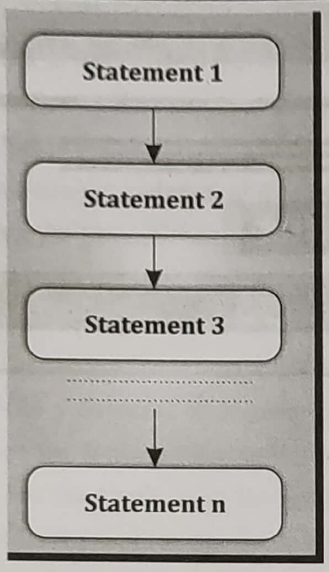
Control Structures

Chapter Outline

- Introduction
- Control Statements
- Decision Making Control Statements
  - Simple If Statement
  - If Else Statement
  - Nesting of If Else Statements
  - The else-if Ladder
  - Switch Statement
- Loop Control Structures
  - while Loop
  - do-while Loop
  - for Loop
- Jumps in Loops
  - The break keyword
  - The Continue Keyword
  - Goto Statement and Labels
- Nested Loops
- Review Questions

7.1 Introduction

So far we have seen C-programs which contains some statements and each statement in a program is executed one by one in a sequence. Here, sequence means the order in which they appear in a program. In this, neither the statement are repeated nor in the order of execution changed as shown in below figure.

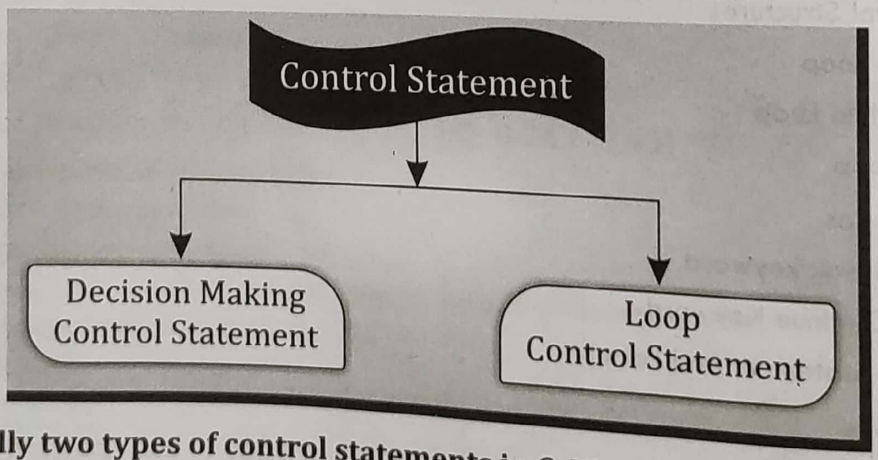


The execution of a C program is top down i.e., execution starts with the beginning of the main() function and executes statement by statement until the end of the main () is reached.

In most of the scenarios, we also need to be able to specify that a statement, or a group of statements, is to be carried out conditionally, only if some condition is true. Also we need to be able to carry out a statement (or) a group of statements repeatedly based on certain conditions. These kind of situations are handled by using **control instructions** (or) **control statements** (or) **control structures.**

7.2 Control Statements

The control statements are used to control the **"flow of control"** in a program. The control statements specifies the order in which the various instructions/statements in a program should be executed. That means it determines the order in which the statements are executed.



There are basically two types of control statements in C. They are

1. **Decision Making Control Statements:**

   The decision making control statements allow the computer to take a decision as to which statement is to be executed next based on certain conditions.

2. **Loop Control Statements:**

   The loop control statements are used to execute a group of statements repeatedly.

7.3 Decision Making Control Statements

In certain situations, it is necessary to use some conditions to make a decision. Based on decision we can execute only a list of statements. The **control statements** are used to make a **decision.**

In many applications, it is required to select a block of statements out of alternatives for execution depending on decision making. If the condition is true, then we execute some block of statements; otherwise we can execute some other block of statements. The condition is normally a comparison between expressions. Several control statements available with C are listed below.

1. **if statement**
2. **if else statement**
3. **nested if statement**
4. **else if ladder**
5. **switch statement**

All the above statements are **control statements** or **decision making statements** or **selection control statements.** We study the above statements in the following sections.

7.3.1 Simple If Statement

In some situations, it may be necessary to carry some operations if the condition is only true. In such cases the simple if statement can be used. Simple **if** is a **one-way branching statement.** When the condition of **if** is **true**, the statements present within the **if block** are executed; otherwise the **if block** is skipped by transferring the control directly to the first statement after the **if block.**

The syntax of if statement begins with keyword **if** followed by the test expression/condition enclosed in a pair of parenthesis. This is followed by a statement (block) as shown below.

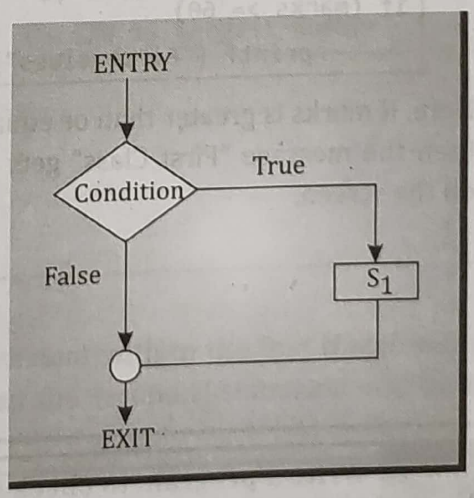
Syntax

```
if (condition)
{
    S1;
}
```

Where

- S_1 may be simple statement (or) compound statement.
- Condition may be test expression which evaluates to either True or False.
- If the condition is evaluated to **true**, then the statements in block S_1 will be executed.
- If the condition is evaluated to **false**, then control is transferred to the succeeding statement after the if-block.

Flowchart



- Only one statement within a block is **simple statement,** on the other hand two or more statements placed within a block make a **compound statement.** If there is only one statement within if-block, then no need of putting statement in braces. i.e., for a simple statement,

Example 1

```
if (a==b)
{
        printf ("%d", a+b);
        printf ("%d", a*b);
        printf ("%d", a/b);
}
```

No semicolon here

- The condition can be
  1. Arithmetic Expression — **Example:** if (a+5)
  2. Relational Expression — **Example:** if (a>5)
  3. Logical Expression — **Example:** if (a>5 && b<5)

Consider an arithmetic expression as a condition as shown below.

```
int a = 10;
if (a+5)
    printf ("\n Inside if");
```

Here, a+5 is an arithmetic expression and if result of a+5 is non-zero number, then condition is evaluated to **true** and hence the statement following if statement will be executed.

In the above case, a+5 = 10+5 = 15 which is non-zero number and all non-zero numbers are true. Suppose if a = –5, then – 5+5 becomes 0 and condition evaluates to **false.**

- There is no restriction on the type of statements to be used in **if-block.** Therefore an **if** statement can have another **if-statement,** loops, etc

7.3.2 If Else Statement

We may need to carry some actions based on **true** or **false** result of the condition. In such situations **if-else** statement is used. Since the actions will be performed for both true and false results of the condition, it is called **two-way branching statement.**

If the condition is true, then the statements present within the **if block** are executed; otherwise the **else block** are executed. The syntax of **if else** statement consists of keyword **if** followed by the condition enclosed in a pair of parenthesis and following which are two statements (or blocks) separated by the keyword **else** as shown below.

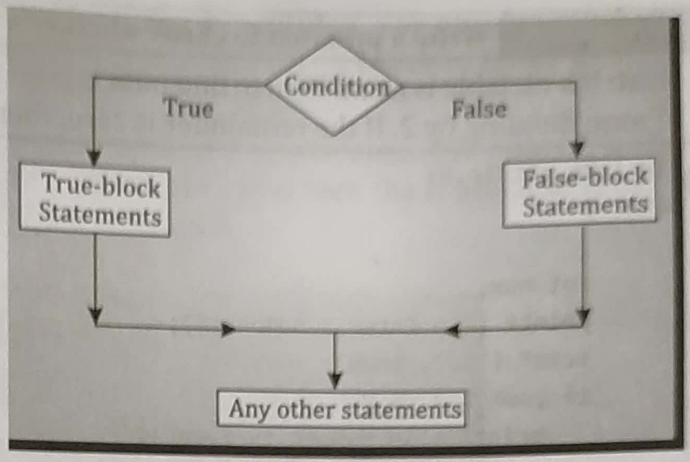
Syntax

```
if (condition)
{
    True-block statement(s);      } ← if block

}
else
{
    False-block statement(s);     } ← else block

}
```

If the condition is true, then the true-block statement(s) are executed; otherwise, the false-block statement(s) are executed. In either case, either True-block or False-block will be executed. **True-block** is also called **if-block** and False-block is also called **else-block.** In both the cases, the control is transferred to the first statement after the **else-block.** The flow chart illustrates the same.



Example

Consider the following code

```
if (marks >= 40)
   printf ("You have passed");
else
   printf ("You have failed");
```

If marks is greater than (or) equal to 40, then message **"You have passed"** will be displayed; otherwise, the message **"You have failed"** will be displayed.

Different forms of if-else statements

Braces are not required if only one statement is present in **if** or **else** block.

```
if (condition)
    statement1;
else
    statement2;
```

```
if (condition)
{
    statement1;
    statement2;
}
else
    statement3;
```

```
if (condition)
    statement1;
else
{
    statement2;
    statement3;
}
```

```
if (condition)
{
    statement1;
    statement2;
}
else
{
    statement3;
    statement4;
}
```

Program 4 **Write a program to find largest of two numbers using if-else statement.**

```
#include <stdio.h>
main()
{
    int n1, n2;
    printf("\n Enter two numbers :");
    scanf("%d %d", &n1, &n2);
    if (n1 > n2)
       printf("\n %d is largest number", n1);
    else
       printf("\n %d is largest number", n2);
}
```

Output:

```
Enter two numbers : 20 40
40 is largest number
```

Explanation

In the above case, the condition (20>40) becomes false and therefore it executes the else block. Finally it prints "40 is largest number".

Program 5 Write a program to check whether a number is even or odd.

Hint: If a variable is **num**, then to find num is even (or) odd use **num%2.** Which gives the remainder of num dividing by 2. If the remainder is zero, then **num** is even; otherwise it is odd.

```
#include <stdio.h>
main( )
{
        int num;
        printf ("\n Enter a number:");
        scanf ("%d", &num);
        if (num % 2 == 0)
           printf ("\n Number is Even");
        else
           printf ("\n Number is Odd");
}
```

Output:

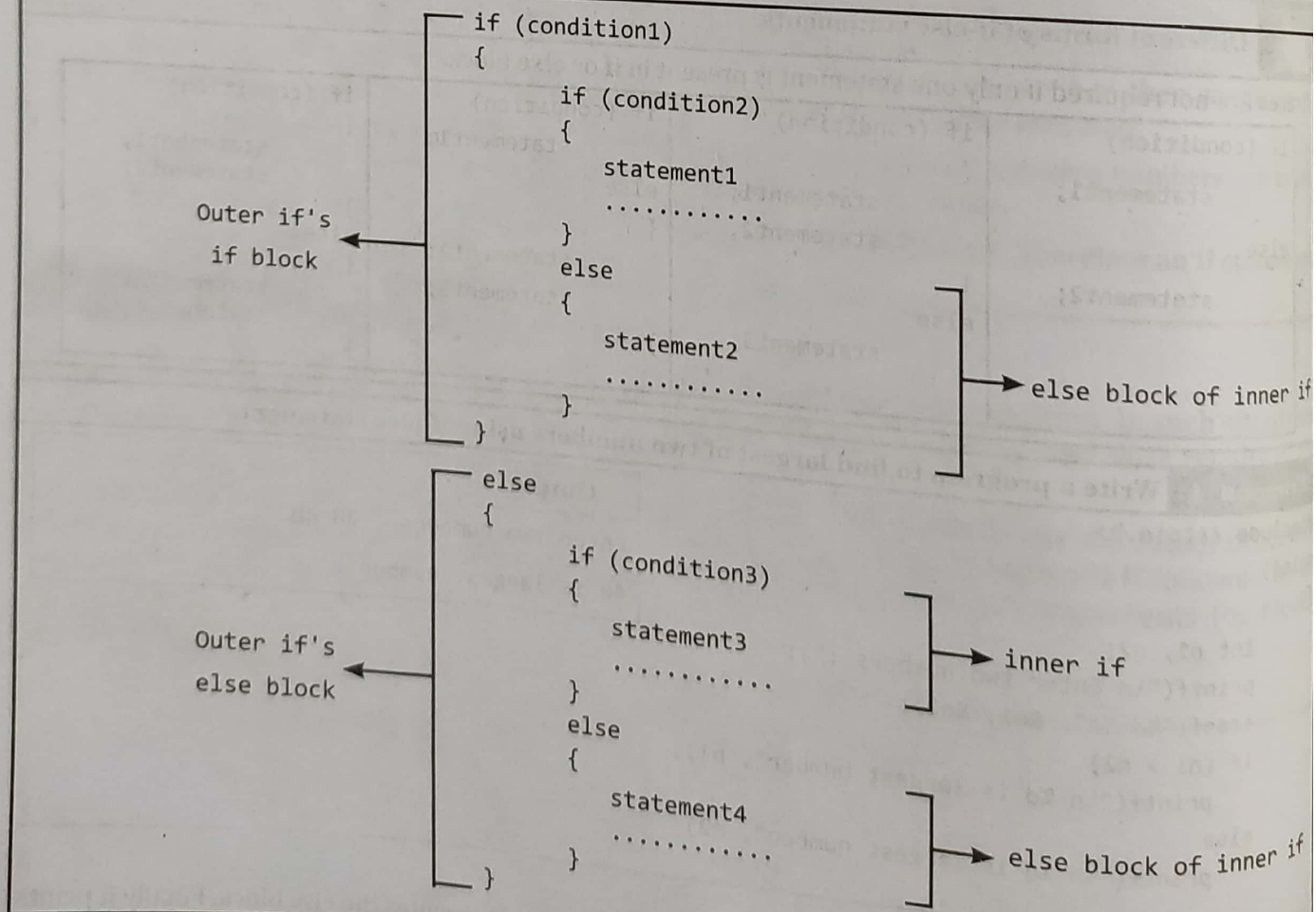
```
Enter a number : 5

Number is Odd
```

7.3.3 Nesting of If Else Statements

A structure consisting of an **if** statement within another if statement (or) else block is referred to as nesting of **if** statements. That means we can write an entire **if construct** (or) **if-else construct** within the body of **if clause** or within the body of **else clause.**

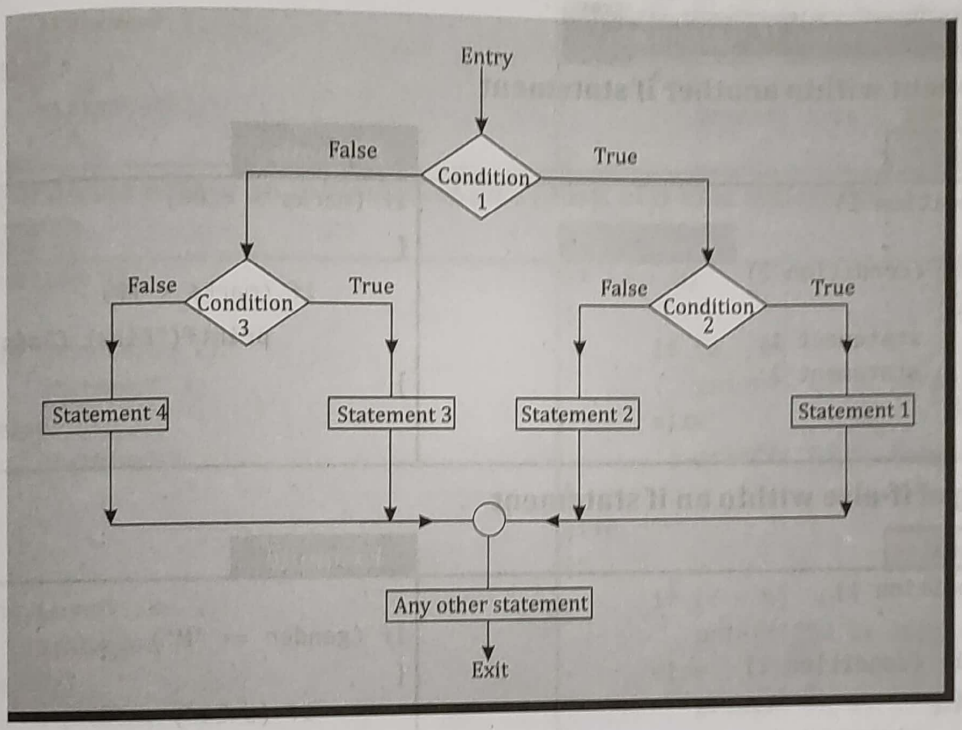
Syntax



When the **outer if** condition is true, the **outer if's if block** will be executed. The **outer if's if block** contains **another if** (i.e., inner if). If the result of **inner if** is true, then the **if block** of **inner if** will be executed; otherwise **else block** of **inner if** will be executed.

When the **outer if** condition is false, the **outer if's else block** will be executed. The **outer if's else block** contains another if (i.e., inner if). If the result of **inner if** is true, then the **if block** of **inner if** will be executed; otherwise **else block** of **inner if** will be executed.

Flow Chart:



Program 6 **Write a program to find largest of three numbers using nested-if statements.**

```
#include <stdio.h>
void main()
{
    int a, b, c;
    printf("\n Enter three numbers :");
    scanf("%d %d %d", &a, &b, &c);
    if (a > b)
    {
       if (a > c)
          printf("\n %d is the largest", a);
       else
          printf ("\n %d is the largest", c);
    }
    else
    {
       if(c > b)
          printf ("\n %d is the largest", c);
       else
          printf ("\n %d is the largest", b);
}   }
```

Output:

```
Enter three numbers : 10 30 20
30 is the largest
```

Explanation

Assume a, b, c are three numbers. First check whether (a>b) is true. If it is true, then check one more condition (a>c) or not. If (a>c) is true, then we can say that **"a is largest".** If (a>c) is false, then we can say that **"c is largest".**

If the first condition (a>b) is false, then it goes to else part to check one more condition (c>b). If (c>b) is true, then we can say that **"c is largest"**. If (c>b) is false, then we can say that **"b is largest"**

Different Forms of Nested If Statements

1. if statement within another if statement

Syntax

```
if (condition 1)
{
      if (condition 2)
      {
         statement 1;
         statement 2;
      }
}
```

Example

```
if (marks > = 60)
{
      if (marks < 70)
         printf("First Class");
}
```

2. Nesting of if-else within an if statement

Syntax

```
if (condition 1)
{
      if (condition 2)
      {
         statement 1;
         statement 2;
      }
      else
      {
         statement 3;
         statement 4;
      }
}
```

Example

```
if (gender == 'M')
{
      if (salary > 5000)
      {
         printf("You have to pay tax");
      }
      else
      {
         printf("No tax");
      }
}
```

3. Nesting of if-else within an if-block of if-else statement.

Syntax

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

Example

```
if (gemder == 'M')
{
      if (salary > 5000)
         printf("You have to pay tax");
      else
         printf("No tax");
}
else
      print("No tax for women");
```

4. Nesting of if-else within an else-block of if-else statement

Syntax

```
if (condition 1)
      statement 1;

else
{
      if (condition 2)
         statement 2;
      else
         statement 3;
}
```

Example

```
if (i < 0)
      print("Negative number");
else
{
      if (i > 100)
         printf("i is greater than 100");
      else
         printf("i is < 100 and > 0 ");
}
```

5. Nesting of if-else within an if-block and else block of if-else statement.

Syntax

```
if (condition 1)
{
      if (condition 2)
         statement 1;
      else
         statement2;
}
else
{
      if (condition 3)
         statement 3;
      else
         statement 4;
}
```

Example

```
if (a > b)
{
      if (a > c)
         printf("%d is largest", a)
      else
         printf("%d is largest", c);
}
else
{
      if (c < b)
         printf("%d is largest", c);
      else
         printf("%d is largest", b);
}
```

Program 7 Write a program to find the smallest of the three numbers.

```
#include <stdio.h>
void main()
{
    int a, b, c, smallest;
    printf("\n Enter three numbers :");
    scanf("%d %d %d", &a, &b, &c);
    if (a < b)
    {
       if (a < c)
          smallest = a;
       else
          smallest = c;
    }
    else
    {
       if (c < b)
          smallest = c;
       else
          smallest = b;
    }
    printf ("\n The smallest = %d", smallest);
}
```

Output:

```
Enter three numbers : 20 10 15
The smallest = 10
```

Program 8 Write a program to calculate the simple interest based on the following conditions.

Principle	Rate of interest
> = 10000	20%
> = 1000 && < = 5000	10%
> 5000 && < 10000	15%

```
#include <stdio.h>
void main()
{
      float principle, years, rate, simpleinterest;
      printf ("\n Enter Principle & No. of years");
      scanf ("%f %f", &principle, &years);
      if (principle > = 10000)
            rate = 20;
      else
      {
         if (principle > = 1000 && principle < = 5000)
            rate = 10;
         else
            rate = 15;
      }
      simpleinterest = (principle * years * rate)/100;
      printf("\n Principle                = %.2f", principle);
      printf("\n Years                    = %.2f", years);
      printf("\n Rate                     = %.2f", rate);
      printf("\n Simple interest          = %.2f", simpleinterest);
}
```

Output:

```
Enter Principle & No. of years : 6000 3
Principle                =   6000.00
Years                    =   3.00
Rate                     =   15.00
Simple interest          =   2700.00
```

7.3.4 The else-if Ladder

The **else-if** ladder is a **multi-way decision maker** which contains two (or) more **else-if,** from which any one block is executed. The syntax of multiple choice with **else if** is as shown below.

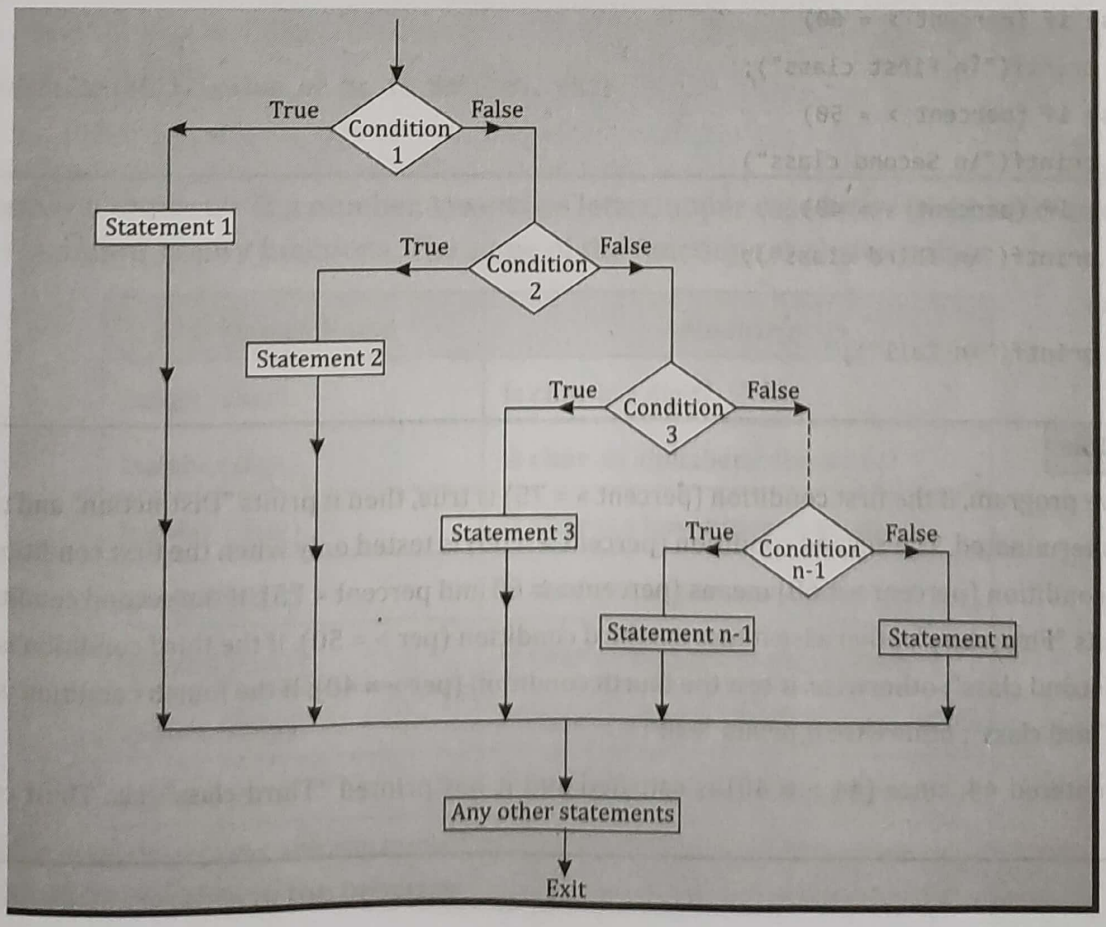
Syntax

```
First if ->      if (condition 1)
                 {
                    statement 1;
                 }
Second if ->     else if (condition 2)
                 {
                    statement 2;
                 }
Third if ->      else if (condition 3)
                 {
                    statement 3;
                 }
                 ..................
                 ..................
```

```
n-1th if ->     else if (condition n-1)
                {
                    statement n-1;
                }
else ->         else
                {
                    statement n;
                    ...........
                }
```

- If the first condition is true, then **statement1** will be executed and control will be sent out of the structure.
- If the first condition is not true, then the second condition, is tested. If the condition of second if is true, then **statement2** will be executed and control will be sent out of the structure.
- If the second condition is not true, then the third condition is tested. If the third condition is true, then **statement3** will be executed and control will be sent out of the structure.
- This process continues as long as the else-if's are present.
- If none of the conditions are true, then **statements** in the **else** block will be executed.
- The **else-block** at the end of the structure is optional.
- If else-block is present, then at least one **statement** or **block** will definitely be executed.
- If else-block is not present and none of the conditions are true, then none of the statements are executed.

Flow Chart:



Program 9 Write a program to accept percentage of a student marks. Depending on the following conditions, print the category of class achieved.

Condition	Category
per > = 75	Distinction
per > = 60	First class
per > = 50	Second class
per > = 40	Third class
otherwise	Fail

```
#include <stdio.h>
void main()
{
        float percent;

        printf("\n Enter the percentage :");
        scanf("&f", &percent);
        if (percent > = 75)
            printf("\n Distinction");
        else if (percent > = 60)
            printf("\n First class");
        else if (percent > = 50)
            printf("\n Second class")
        else if (percent > = 40)
            printf("\n Third class");
        else
            printf("\n Fail");
}
```

Output:

```
Enter the percentage : 44
Third class
Enter the percentage : 90
Distinction
```

Explanation

In the above program, if the first condition (percent > = 75) is true, then it prints "Distinction" and thereby the program is terminated. The second condition (percent > = 60) is tested only when the first condition becomes **false.** The condition (percent > = 60) means (percent > = 60 and percent < 75). If the second condition is true then it prints "First class"; otherwise it test the third condition (per > = 50). If the third condition is true, then it prints "Second class"; otherwise, it test the fourth condition (per > = 40). If the fourth condition is true, then it prints "Third class"; otherwise it prints "Fail".

When we entered 44, since (44 > = 40) is satisfied and it has printed "Third class". i.e., Third condition is satisfied.

Program 10 **Write a program to find whether a character accepted is a numeric, lower case letter, upper case letter or a special character. Print the ASCII value of accepted character.**

We know that every character has an ASCII value and which gets printed by using %d format specifier. Because ASCII value is an integer type.

```
#include <stdio.h>
main()
{
    char ch;
    printf("\n Enter a character :");
    scanf("%c", &ch);
    if (ch > = '0' && ch < = '9')
       printf("\n Numeric character");
    else if (ch > = 'a' && ch <= 'z').
       printf("\n Lower case character");
    else if (ch > = 'A' && ch < = 'Z')
       printf("\n Upper case character");
    else
       printf("\n Special character");

    printf("\n ASCII value of %c is %d", ch, ch);
}
```

Output:

```
Enter a character : 4
Numeric character
ASCII value of 4 is 52

Enter a character : A
Upper case character
ASCII value of A is 65
```

To find whether a character is a number, lowercase letter, upper case letter (or) blank space, we can also use C – standard library functions. The some of the functions are listed below.

Function Name	Meaning
isdigit (char)	Is **char** is a digit?
isalpha (char)	Is **char** an alphabetic character?
islower (char)	Is **char** is a lower case character?
isupper (char)	Is **char** is a upper case character?
isalnum (char)	Is **char** an alphanumeric character?
isspace (char)	Is **char** is a white space character?
isprint (char)	Is **char** is a printable character?

The above character functions are contained in the file **ctype.h** and therefore the statement #include **<ctype.h>** must be included in the program.

Program 11 Write a program to find whether a character accepted is a number, lower case letter, upper case letter, or a special character using C-Standard Library Functions.

```
#include <stdio.h>
#include <ctype.h>
main()
{
      char ch;
      printf("\n Enter a character :");
      scanf ("%c", &ch);
      if (isdigit(ch))
         printf("\n Numeric character");
      else if (islower(ch))
         printf("\n Lowercase character");
      else if (isupper(ch))
         printf("\n Uppercase character");
      else
         printf("\n Special character");
}
```

Output:

```
Enter a character : 8
Numeric character

Enter a character : a
Lowercase character
```

7.3.5 Switch Statement

The **switch** statement provides multi-way branching (similar to else-if ladder). The switch is a extension of **if-else** structure. Using if-else a maximum of two branches are allowed. If there is need for multiple branches, **nested-if** can be used or **else-if ladder** can be used. Instead, a **switc statement** can be used which is a better choice over **nested-if** and **else-if ladder**. The **switc** statement is used whenever multiple branches are required.

The switch allows the user to select any one of the alternatives depending on the value of a **expression.** The **expression** present within a parenthesis of **switch** statement must be of the ty **int** (or) **char**. Based on the expression, the control is transferred to a particular **case label** and th the statements associated with that **case label** are executed. The **case label** must be of the ty **int (or) char.** The case **default** is executed only when none of the cases are true to the value of t expression present in **switch** construct.

Syntax

```
switch (expression)
{
      case label 1:   statement 1;
                      ..............
                      break;
      case label 2 :  statement 2;
                      ..............
                      break;
      case label 3:   statement 3;
                      ..............
                      break;
      ....................................
      ....................................
      case label n :  statement n;
                      .........
                      break;
      default :       default statements;
                      ...................
}
statement-x;
```

- The expression following the keyword **switch** is evaluated and which results in a **constant.**
- This constant is compared with each case label in sequentially until match occurs.
- If the expression value is same as case label, then the statements corresponding to that particular **case** are executed. Subsequently, the execution of **break** statement sends the control out of the switch-construct. Which means control reaches **statement-x;**
- In case the expression value does not match with any case label, the statements in the **default** block will be executed. The presence of **default** block is optional.
- The **expression** should be of integer type. That is it can contain variables and constants of type **int** and **char.** Since each character has an equivalent integer value.

Example 1

1.	`int ch = 2;` `switch (ch)`	→ valid
2.	`char ch = 'A';` `switch (ch)`	→ valid
3.	`switch (2+(3*4))` Here, 2+(3*4) is an integer expression.	→ is valid
4.	`switch (3.5+2.4)` is invalid	

- Label after each **case** has to be a constant or constant expression of integer type.

Example 2

```
case 4 :
case 'A':          → valid
case 4*2 :
case (2+3)*4 :

case 2.5 :
case "str" :       → invalid
case 2.5+10 :
```

- No two labels can have same values.
- The **labels** with several cases need not be in order.
- More than one statement with a **case** need not be enclosed in braces.

Example 3

```
case 1 :{
            statement 1
            statement 2,
            break,
        }
```

Braces Not required

- It is not compulsory to have the statements for a particular case.

Example 4

```
case 1 :
case 2;
case 3 :   statement;
           break;
```

- There should not be a semicolon at the closing parenthesis of expression and at the closing braces.

```
switch(expression)
{
    case 1:
        .
        .
        .
    default
}
```

No semicolon

No semicolon

- Nesting of **switch** structure within the case of another switch structure is allowed.

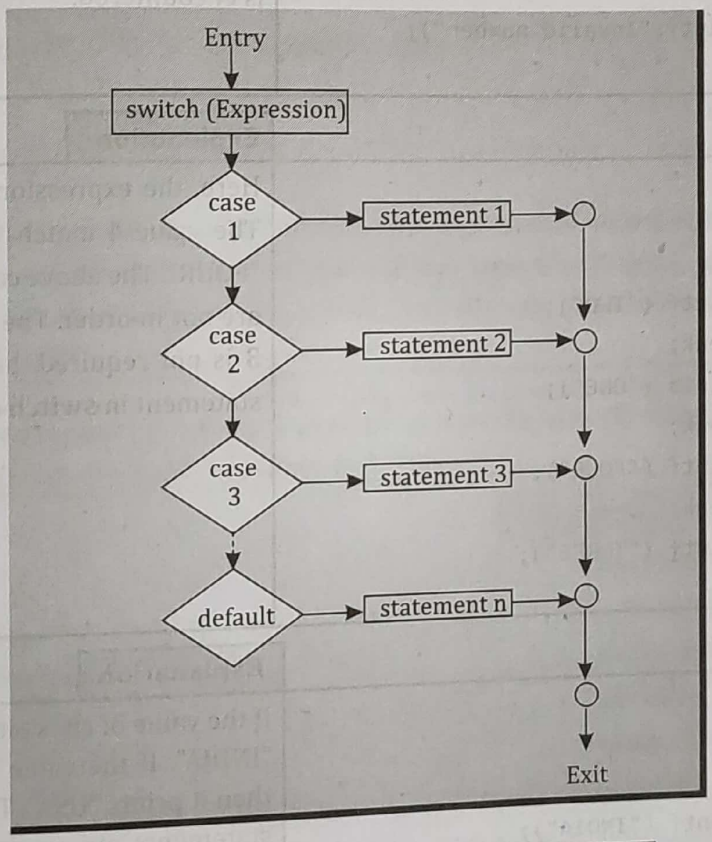
Example 5

```
switch (ch)
{
      case 1:   switch (a)
                {
                      case 10 :   .....
                      case 11 :   .....
                }
      case 2:   .....
                break;
      case 3:   .....
                break;
}
```

- The **default** block can be placed any where in the structure. But it is generally placed at the end of structure.
- The character constants used in the case label are automatically converted into integers (i.e., their ASCII value equivalent)

$$\text{case 'A':} \xrightarrow[\text{as}]{\text{same}} \text{case 65:}$$

- **break** is a keyword and used to terminate the **switch construct.** The control is transferred to the first statement after the **switch** construct.



Example 6

```
int a = 2;
switch (a)
{
    case 1:  printf ("Number is 1");
             break;
    case 2:  printf("Number is 2");
             break;
    case 3:  printf("Number is 3");
             break;
    default: printf("Invalid number");
}
```

Explanation

Here, the value of a is 2 and it match with case label 2. Therefore it prints **"Number is 2".** If the value of a is 3, then it prints "Number is 3". If the value of a is 4, then none of the case labels are matching with 4 and therefore it goes to **default** and prints "Invalid number".

Example 7

```
int a = 2;
switch (a)
{
   case 1:   printf("Number is 1");
             break;
   case 2:   printf("Number is 2");
   case 3:   printf("Number is 3");
   default:  printf("Invalid number");
}
```

Explanation

The above example generates the output

```
Number is 2
Number is 3
Invalid number
```

The reason is we have not used **break** after the case label 2. It prints all subsequent statements until it encounters break or default is encountered.

Example 8

```
int a = 3;
switch (a+1)
{
   case 2:   printf ("TWO");
             break;
   case 1:   printf ("ONE");
             break;
   case 4:   printf ("FOUR");
             break;
   case 3:   printf ("THREE");
}
```

Explanation

Here, the expression (a+1) is evaluated to 4. The value 4 match with case 4 and it prints "FOUR". The above code shows that case labels are not in order. The **break** at the end of **case 3** is not required, because **case 3** is the last statement in **switch** construct.

Example 9

```
switch (ch)
{
   case 'i':
   case 'I': printf ("INDIA");
             break;
   case 'u':
   case 'U':    printf ("USA");
}
```

Explanation

If the value of ch is either 'i' or 'I', then it prints "INDIA". If the value of ch is either 'u' or 'U', then it prints "USA". This code clearly tells that statements are not compulsory for any case label.

Program 12 **Write a program to check whether the input character is vowel (or) not.**

```
#include <stdio.h>
void main()
{
      char ch;
      printf("\n Enter a character:");
      scanf("%c", &ch);
      switch (ch)
      {
```

Output:

```
Enter a character : I
I is vowel
Enter a character : B
B is not a vowel
```

```
        case 'A' :
        case 'a' :
        case 'E' :
        case 'e' :
        case 'I' :
        case 'i' :
        case 'O' :
        case 'o' :
        case 'U' :
        case 'u': printf("%c is vowel", ch);
                  break;
        default : printf ("%c is not a vowel", ch);
    }
}
```

Explanation

In the above program, the label ('A', 'a', ... 'U') of each **case** is compared with the variable value **ch**. If the **case** label is equal to the variable **ch**, then the statements of that **case** are executed until the **break** statement is encountered. If none of the case labels match with **ch**, then **default** will be executed and prints the message as **"character is not a vowel".**

Program 13 **Write a program to perform the operations (1) Add, (2) Subtract, (3) Multiplication, (4) Division, (5) Largest of two numbers**

```
#include <stdio.h>
void main()
{
    int a, b, c, ch;
    printf("\n 1.Addition");
    printf("\n 2.Subtraction");
    printf("\n 3.Multiplication");
    printf("\n 4.Division");
    printf("\n 5.Largest of two numbers");
    printf("Enter your choice :");
    scanf("%d", &ch);
    if (ch < = 5 && ch > 0)
    {
        printf ("Enter two numbers :");
        scanf ("%d %d", &a, &b);
    }
    switch (ch)
    {
        case 1:   c = a+b;
                  printf ("\n Addition: %d", c);
                  break;
        case 2:   c = a-b;
                  printf ('\n Subtraction: %d", c);
                  break;
```

Output:

```
1. Addition
2. Subtraction
3. Multiplication
4. Division
5. Largest of two numbers
Enter your choice : 5
Enter two numbers : 8 9
b is larger.
Enter your choice : 4
Enter two numbers : 20 10
Division : 2
```

```
        case 3:  c = a*b;
                 printf ("\n Multiplication: %d", c);
                 break;
        case 4:  c=a/b;
                 printf ("\n Division: %d", c);
                 break;
        case 5:  if (a>b)
                    printf ("\n a is larger");
                 else if (b>a)
                    printf ("\n b is larger");
                 else
                    printf ("Both a & b are same");
                 break;
        default: printf ("Invalid choice");
    }
}
```

What are the two different ways to implement a multi-way selection in C ?
else-if ladder and switch
Can we use string value/variable in switch test condition?
No, switch statement works with only integer type of variables/literals like integer, short, character.
Can we have duplicate case values in switch?
Duplicate case values are not allowed.

7.3.6 Conditional Operator (? :)

We have already studied the conditional operator in previous chapter. The conditional operator (or) ternary operator is an alternative for **if-else statement.** The general form of if is

```
if (condition)
        statement1;
else
        statement2;
```

is same as

```
condition ? statement1; statement2;
```

if-else statement **using ternary operator**

Example 1

```
if (number % 2 == 0)
        printf("Even");
else
        printf("Odd")
```

is same as

```
(number%2 == 0) ? printf("Even"):printf("Odd");
```

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

is same as

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

Example 2 The following code is to find the largest of three numbers.

```
if (a>b)
{
   if (a>c)
      largest = a;
   else
      largest = c;
}
else
{
   if (c>b)
      largest = c
   else
      largest = b
}
```

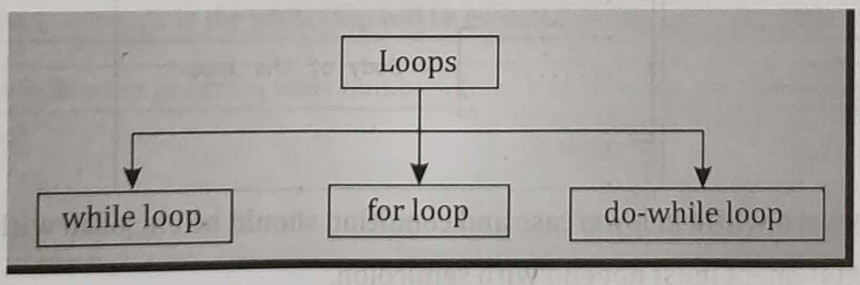
The above code can be written using ternary operator in one line as shown below.

```
largest = (a > b) ? (a > c ? a : c) : (c > b ? c : b);
```

7.4 Loop Control Structures

So far, we have studied the different decision making control statements such as if, if-else, nested-if, switch etc. These statements are executed only once during the execution of the program. In some situations, it may be necessary to execute a list of statements more than once. In such cases, the **looping constructs** are used.

Looping costruct is a construct that executes the statements repeatedly for a certain number of times as long as the condition is **true.** The following are three looping constructs available with C. The **looping constructs** is also called as **repetitive structures.**



Let us study all these looping constructs in detail. The following sections covers the entire concepts of loops.

7.4.1 while Loop

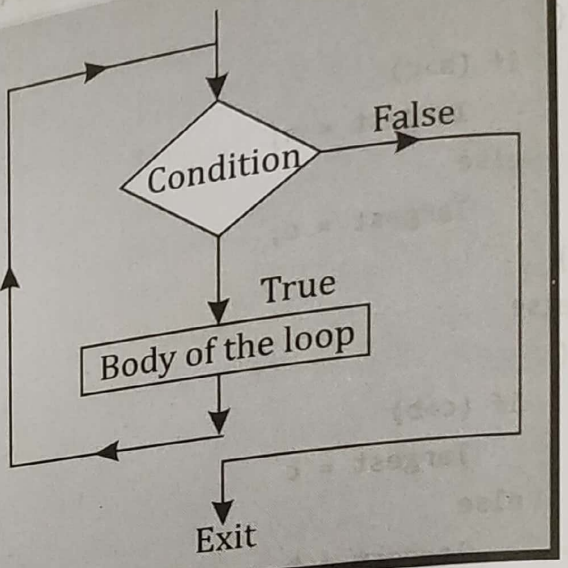
While is a looping construct, used to execute the statements repeatedly as long as the condition is **true.** The syntax of **while** statement with the condition enclosed in a pair of parenthesis and which is followed by the statements as shown below. **while** loop is also called as **"pre-tested"** looping construct (or) **"entry controlled loop".**

Syntax:

```
while (condition)
{
    statement 1;
    statement 2;
    .....
    .....
    statement n;
}
```

Body of the loop

Flowchart:



- The condition is basically test expression, which is evaluated to either **TRUE or FALSE.**
- If the value of condition is true, then it executes all the statements in the body of loop after which the control is transferred to the while statement for evaluation of condition again.
- Thus body of the while loop is executed repeatedly as long as the condition is true.
- The execution of loop is terminated, when the condition is evaluated to FALSE. Then the control is transferred to the first statement after while loop.
- While loop is called as **entry-controlled loop** because the condition is tested in the beginning of while loop.
- The body of loop will not be executed at all, if the condition is FALSE for the first time itself.

```
while (condition)
{
.......
.......        Body of the loop
.......
}
```

- Use the keyword **while** in lowercase and condition should be enclosed within parenthesis.
- The while statement must not end with semicolon.

```
while (condition)          <-- No semicolon
{
 ............
 ............
}
                           <-- No semicolon
```

- The curly braces are optional when the body of the loop contains only one statement. If more than one statement, enclose the statements in curly braces.

```
while(i < 5)
i++;
```

```
while(i<5)
{
    print("%d", i);
    i++;
}
```

Example 1

Consider the following code which prints the numbers 1 to 5.

```
i = 1;
while(i<=5)
{
    printf("%d", i);
    i++;
}
```

Explanation

Initially the variable 'i' is assigned with 1. During the first execution of while loop, the condition (i <= 5) is tested first. Since (1 <= 5) is true, it prints the value of i (i = 1) on the screen and i value gets incremented by 1 (i = 2).

Now again the control is transferred back to the while loop condition. The current value of i is 2 and (2<=5) becomes true. Since the condition is true, the body of the loop is executed again. This process continues as long as the condition is true. Finally, when the value off 'i' becomes 6, the loop is terminated because (6 <= 5) becomes false. All the statements in the while loop will be executed as long as i is less than or equal to 5.

Program 14 **The following program adds numbers from 1 to 10 using the while loop.**

```
#include <stdio.h>
void main()
{
    int i = 1, sum = 0;
    while (i <= 10)
    {
        printf("%d", i);
        sum = sum+i;
        i++;
    }
    printf("\n Sum of first 10 numbers = %d", sum);
}
```

Output:

```
1 2 3 4 5 6 7 8 9 10
Sum of first 10 numbers = 55
```

Explanation

The variable 'i' is initialized to 1 and variables 'sum' is to 0. The while loop checks the condition (i <= 10). The variable 'i' is added to variable 'sum' and the value of 'i' is increased by 1. In each while loop, 'i' is increased and added to 'sum'. When the value of 'i' reaches 11, the condition in while loop becomes false. Then the loop is terminated and control goes to the printf( ) statement after while and prints the sum of 10 numbers. The steps performed are shown below.

i = 1, sum = 0

Step	while (i < = 10)	printf("%d", i);	sum = sum+i	i++
1	while (1< = 10) = True	Prints 1	sum = 0+1= 1	i=2
2	while (2 < = 10) = True	Prints 2	sum = 1+2 = 3	i=3
3	while (3 < = 10) = True	Prints 3	sum = 3+3 = 6	i=4
4	while (4< = 10) = True	Prints 4	sum = 6+4 = 10	i=5
5	while (5 < = 10) = True	Prints 5	sum = 10+5 = 15	i=6
6	while (6 < = 10) = True	Prints 6	sum = 15+6 = 21	i=7
7	while (7 < = 10) = True	Prints 7	sum = 21+7 = 28	i=8
8	while (8 < = 10) = True	Prints 8	sum = 28+8 = 36	i=9
9	while (9 < = 10) = True	Prints 9	sum = 36+9 = 45	i=10
10	while (10 < = 10) = True	Prints 10	sum = 45+10 = 55	i=11
11	while (11 < = 10) = False			

Program 15 **Write a program to calculate the factorial of a given number using while loop.**

If 'n' is a number, then factorial of n is n! = 1* 2 * 3 - - - - * n

if n = 3, 3! = 1* 2 * 3 = 6

if n = 4, 4! = 1 * 2 * 3 * 4 = 24

if n = 5, 5! = 1 * 2 * 3 * 4 * 5 = 120.

```
#include <stdio.h>
void main()
{
    int n, fact = 1;
    printf("\n Enter the number:");
    scanf("%d", &n):
    while (n >= 1)
    {
       fact = fact * n;
       n--;
    }
    printf("\n Factorial of given number is %d", fact);
}
```

Output:

```
Enter the number: 4
Factorial of given number is 24
```

Explanation

We know that factorial of a number means product from 1 to that number. The variable **'fact'** is initialized to 1. For each iteration of **while** loop the **fact** is multiplied with **n** and 'n' value gets decremented by 1 at each iteration. When 'n' value becomes '0', then loop terminates and prints the factorial of given number.

Initially fact = 1 and Assume n = 4

Iteration	while (n > = 1)	fact = fact * n	n--
1	while (4 >= 1) = True	fact = 1 * 4 = 4	n=3
2	while (3 >= 1) = True	fact = 4 * 3 = 12	n=2
3	while (2 >= 1) = True	fact = 12 * 2 = 24	n=1
4	while (1 >= 1) = True	fact = 24 * 1 = 24	n=0
5	while (0 >= 1) = False	--	--

Program 16 **Write a program to reverse a number.**

```
#include <stdio.h>
void main()
{
      long rev, num;
      int digit
      printf("\n Enter the number:");
      scanf("%ld", &num);
      rev = 0;
      while (num>0);
      {
         digit = num %10;
         num = num/10;
         rev = rev*10+digit;
      }
      printf("\n The Reverse of a given number = %ld", rev);
}
```

Output:

```
Enter the number: 1234
The Reverse of a given number = 4321
```

Explanation

The statement digit = num % 10 gives the remainder and num = num/10 gives the quotient. The iterations performed are shown in the following table. Initially assume num = 1234 and rev = 0.

Iteration	while (num > 0)	digit = num %10	num = num/10	rev = rev * 10+digit
1	while (1234>0) = True	digit = 1234 %10 digit = 4	num = 1234/10 num = 123	rev = 0 * 10+4 rev = 4
2.	while (123>0) = True	digit = 123 %10 digit = 3	num = 123/10 num = 12	rev = 4 * 10+3 rev = 40+3 = 43
3	while (12>0) = True	digit = 12% 10 digit = 2	num = 12/10 num = 1	rev = 43 * 10+2 rev = 430+2 = 432
4	while (1> 0) = True	digit = 1 %10 digit = 1	num = 1/10 num = 0	rev = 432 * 10+1 rev = 4320+1 = 4321
5	while (0 >0) = False	—	—	—

The last value of **rev** is the reverse of a given number **1234**. Print this number using **printf( )** statement.

Program 17 **Write a program to find sum of digits of given number, using while loop.**

Suppose n = 1234, then sum of the digits of the number 1234 is 1+2+3+4=10. Find the last digit of number using digit = num % 10 and add this digit to the variable **sum**. Repeat this process until num becomes Zero.

```
#include <stdio.h>
void main()
{
        int num, digit, sum = 0;
        printf("\n Enter the number:");
        scanf("%d", &num);
        while (num > 0)
        {
            digit = num % 10;
            sum = sum + digit;
            num = num/10;
        }
        printf ("\n The sum of digits of given number is %d", sum);
}
```

Output:

```
Enter the number : 1234
The sum of digits of given number is 10
```

Explanation

Assume n = 1234, sum = 0. The each iteration is explained as shown below.

Iteration	while (num)0)	digit = num %10	sum = sum+digit	num = num 10
1.	while (1234 > 0)	digit = 1234 %10 digit = 4	sum = 0+4 sum = 4	num = 1234/10 num = 123
2.	while (123>0) = True	digit = 123%10 digit = 3	sum = 4+3 sum = 7	num = 123/10 num = 12
3.	while (12>0) = True	digit = 12%10 digit = 2	sum = 7+2 sum = 9	num = 12/10 num = 1
4	while (1>0) = True	digit = 1%10 digit = 1	sum = 9+1 sum = 10	num = 1/10 num = 0
5	while (0>0) = False	—	—	—

The last value of **sum = 10** is printed using the **printf( )** statement.

Example 2

Consider the following code

```
a = 10;
while (a)
{
        printf("%d", a):
        a--;
}
```

The variable 'a' is non-zero and hence while loop gets executed. If the variable 'a' becomes 0, then loop gets terminated. Any non-zero value is considered as TRUE and zero is considered as FALSE. The above code prints the values from 10 to 1. Once the value of 'a' becomes 0; then the condition becomes false. Observe the following code.

```
while (0)
{
        printf("Hello");
        printf("Hai");
}
```

The constant '0' indicates FALSE, So, the body of the loop will never get executed. The control will not be transferred inside the loop.

```
while(1)
{
        printf("Hello");
        printf("Hai");
}
```

Here, the constant '1' is TRUE, So, the body of the loop will be executed infinite number of times. Only way to terminate the loop is by using **break / return** based on some condition within the loop. Consider the following code carefully.

```
a = 3;
while (a <= 5);
{
        a++;
}
```

There is a semicolon (;) at the end of while. It means as long as (a <= 5) do not do anything. It waits infinitely till the above condition becomes false which will never happen. It is an example of an **infinite loop.**

7.4.2 do-while Loop

In **while** loop, the condition is tested in the beginning. If the condition becomes false for the first time, then the body of **while** loop will not be executed even once. In some programming situations, we may require to execute the body of a loop at least once even though the condition is false for the first time. In such situations, **do..while** is used. This structure is also called as the **"Post-tested"** looping construct or **"exit-controlled loop".**

The condition is tested after executing the body of the loop for first time. If the condition is **true,** then body of the loop will be executed again. If the condition is **false,** then the control is transferred out of the **do..while** construct. i.e., the body of the loop will be executed again and again until the condition becomes **FALSE.**

The syntax of **do..while** is shown below.

The **body of the loop** is executed first. At the end of the **do..while** loop, the condition is tested. If this condition results to true, then the **body of the loop** is executed once again. This process continues as long as the condition is **true.** When the condition becomes **false**, the loop will be terminated and control reaches the **statement -x.**

The semicolon is required at the end of **while**.

```
do
{
        printf("%d", i);
        i--;
} while (i > 0);.
```

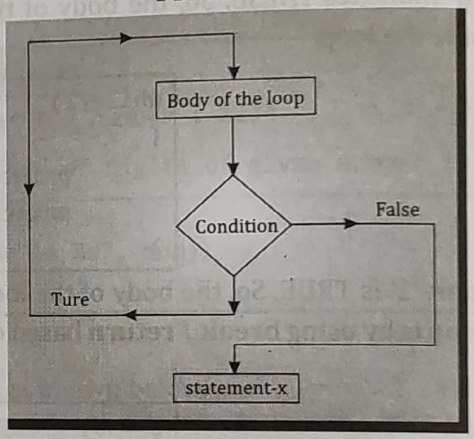
← Semicolon required

Syntax

```
do
{
        statement 1
        statement 2
        ..........
        ..........
}
while (condition);

statement-x;
```

Flowchart



Program 18 **Consider the following program which prints the numbers from 1 to 5.**

```
#include <stdio.h>
void main()
{
      int i = 1;
      do
      {
         printf("%2d", i);
         i++;
      } while (i <= 5);
}
```

Output:

```
1 2 3 4 5
```

Explanation

The program first initializes the value of the variable 'i' to 1. Then control reaches to **printf( )** statement and which prints the value of i (prints 1), then the value of i becomes 2. Now the condition is tested in the while and (2 <= 5) becomes true. This process is continued until the value of **'i'** becomes 6 and (6 <= 5) becomes false, so the loop is terminated. The iterations are showed in the following table. Initially i = 1;

Iteration	printf ("%2d", i)	i ++	while (i<= 5)
1	Prints 1	i=2	while (2 <= 5) = True
2	Prints 2	i=3	while (3 <= 5) = True
3	Prints 3	i=4	while (4 <= 5) = True
4	Prints 4	i=5	while (5 <= 5) = True
5	Prints 5	i=6	while (6 <= 5) = False

Program 19 The following program demonstrates the execution of statements in do..while exactly once.

```
#include <stdio.h>
void main()
{
    int i = 10;
    do
    {
       printf("%d", i);
       i++;
    } while (i <= 5);
}
```

Output:

```
10
```

Explanation

We can observe that the statements within the loop executed for the first time even though the condition is false. First it has printed the value of i = 10 and value of i has incremented by 1. i.e., i = 11. Then it has tested the condition (i <= 5) and i.e., 11 <= 5 which is false and loop is terminated.

Program 20 The following program accepts a number from the user, displays digits and finds sum of digits in the number.

```
#include <stdio.h>
void main()
{
    int num, digit, sum = 0;
    printf("\n Enter a number :");
    scanf("%d", &num);
    do
    {
       digit = num %10;
       printf("%d", digit);
       sum = sum+digit;
       num = num/10;
    } while (num != 0);

    printf("\n The sum of the digits are %d", sum);
}
```

Output:

```
Enter a number: 345
Digits in the number are... 5 4 3
The sum of the digits are 12
```

Explanation Initially sum = 0, and assume num = 345.

Iteration	digit = num %10	printf("%d",digit)	sum=sum+digit	num=num/10
1	digit = 345 %10 = 5	print '5'	sum = 0+5 = 5	num = 345/10 = 34
2	digit = 34 %10 = 4	prints '4'	sum = 5+4 = 9	num = 34/10 = 3
3	digit = 3 % 10 = 3	prints '3'	sum = 9+3 = 12	num = 3/10 = 0

In the third iteration, the value of **num** becomes **zero** and hence the loop is terminated.

What is the difference between Entry Controlled and Exit Controlled loop?

In Entry Controlled Loop, loop body is checked after checking the test condition i.e condition is checked first after that loop body will execute while in Exit Controlled Loop loop body will be executed first after that loop's test condition is checked.

Example: Entry Controlled Loops are : for, while

Exit Controlled Loop is : do while.

7.4.3 for Loop

The **for** loop is also called as **"Pre-test loop"** or **"Fixed-execution loop"**. The **for** loop is the modifie form of a **while** loop. The **for** loop is especially used to execute the statements for a certain numbe of times. This loop simplifies the program as well as can reduce the number of statements. In a whil loop, before the condition is tested, all the variables used in the condition are initialized. In a **for** loo the variable initialisation can be done in the **for** statement itself. The general syntax of **for** loop i shown below.

Syntax

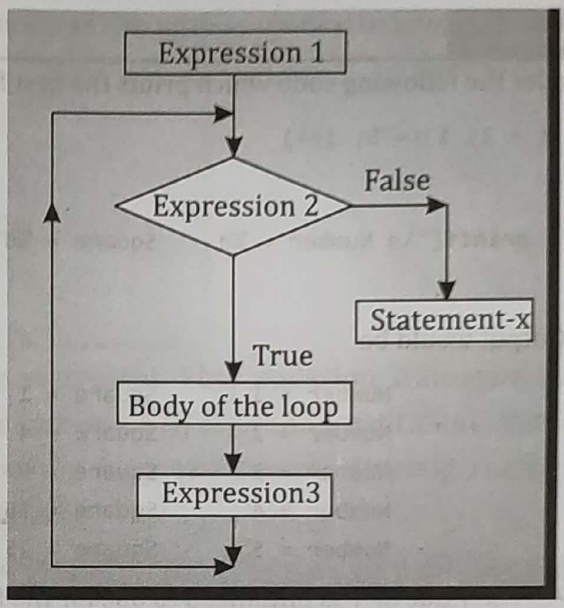
```
for (Expression1; Expression2; Expression3)
{
        statement1;
        statement2;
        ..........
        ..........
}
statement-x;
```

Where

- **Expression 1** is initialization
- **Expression 2** is the conditional expression
- **Expression 3** is an updation. Which may be assignment statement (or) increment expressio (or) decrement expression.
- **Initialization:** The loop variable is initialized with a value when the control enters the fo loop for first time. That means **expression 1** will be executed only once in the beginning.
- **Conditional Expression:** This is generally a relational expression. The for loop is executed as long as the conditional expression is True.
- **Updation:** Which Updates the value of loop variable after each execution of body of **for** loop Generally it is assignment statement (or) increment (or) decrement expression.

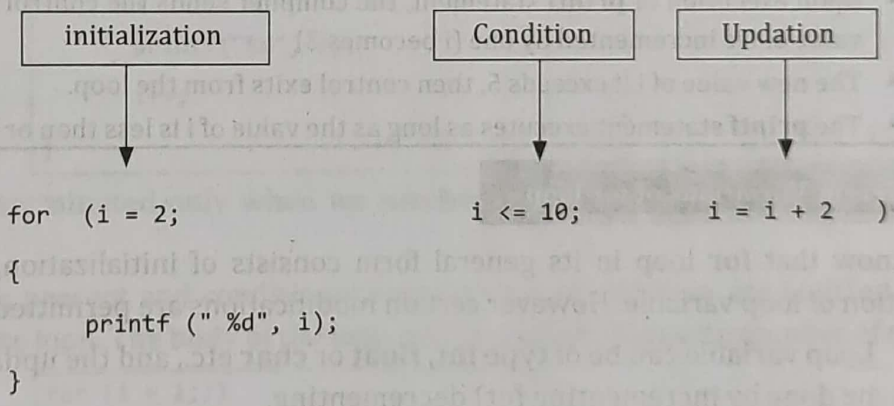
The **expression 1** will be executed only once in the beginning. Then the **expression 2** is evaluated If the **expression 2,** is evaluated to **true**, then body of the for-loop will be executed and then the statements in **expression 3** will be executed.

Usually **expression 3** changes the variables used in the **expression 2.** The **expression 2** is checked again and the process is repeated. The body of the for-loop will be executed as long as **expression 2** is evaluated to true. Once **expression 2** is evaluated to **false,** control goes out of the **for** loop and **statement-x** following for-loop will be executed. The flow chart of for-loop execution is given below.



Example 1

Let us consider a simple task of displaying the even numbers from 2 to 10.



- When the control enters the for statement for the first time, the loop variable i' is initialized to 2. ***Initialization of loop variable is done only once at the beginning of first iteration.***
- Then the control reaches to the conditional expression. The condition (i <= 10) is tested and (2 <= 10) is true, So, the body of loop is executed by printing the value **2.**
- After executing the body of the for loop, the control reaches to the **Updation** statement. Where the value of i is incremented by 2. Now 'i' becomes 4.
- After executing the **Updation** statement, the control reaches to the condition again. Which checks the **condition** (4 <= 10) and it evaluates to true. Then again body of the loop is executed by printing the value of i (i = 4).
- This process is carried out as long as the condition is true. Once the value of 'i' reaches 12, then loop is terminated.
- The above code prints 2 4 6 8 10

Example 2

Consider the following code which prints the first five numbers along with their squares.

```
for (i = 1; i <= 5; i++)
{
      printf("\n Number = %d     Square = %d", i, i * i);
}
```

The Output would be

```
Number = 1      Square = 1
Number = 2      Square = 4
Number = 3      Square = 9
Number = 4      Square = 16
Number = 5      Square = 25
```

- The value of 'i' is initialized to one for the first time when execution starts in the for loop.
- The condition i <= 5 is tested in each iteration. Initially the condition is true since the value of i is 1. Then the statement following the for loop executes.
- The printf statement prints the value of i and square of i (i.e 1 and 1)
- Upon execution of **printf** statement, the compiler sends the control back to the for loop where the value of i is incremented by one (i becomes 2)
- The new value of i if exceeds 5, then control exits from the loop.
- The **printf** statement executes as long as the value of i is less than or equal to 5.

Different Variations of for loop

We know that **for** loop in its general form consists of initialization, conditional expression and updation of loop variable. However certain modifications are permitted to suit different situations.

- Loop variable can be of type **int, float** or **char** etc., and the updation of the loop variable can be done by incrementing (or) decrementing.

```
float f;
for (f = 2.5; f <= 10.5; f = f+1.0)
{
      printf("%f", f);
}
```

- The variables which are initialized in the initialization can be written above the **for** loop as

```
int i;
i = 1;
for (; i <= 5; i ++)
{
      printf("%d", i);
}
```

The initialization statement i = 0 is shifted from **for** loop to a location prior to the statement. However the **semicolon** (;) must precede the **conditional expression.**

- The updation statement can be shifted to the body of the loop as shown below.

```
i = 1;
for (; i <= 5;)
{
        printf("%d", i);
        i++;
}
```

The updation statement is omitted in the for statement. This updation statement can be placed at the end of the body of the **for** loop. The semicolon (;) after the conditional expression (i < = 5) is must. After executing the body of the loop, the control directly reaches the conditional expression for checking the condition.

- If the condition (i < = 5) is omitted, then body of the for-loop will be executed infinite number of times. It will go to infinite loop.

```
i = 1
for (;;)
{
        printf("%d", i);
        i++;
}
```

The for loop will be terminated only when we use **break** or **return** statement based on certain conditions.

- If only initialization is present and conditional expression and updation are omitted, then also it will go to infinite loop. The body of the loop will be executed infinite number of times.

```
for (i = 1;;)
{
        printf("%d", i);
        i++;
}
```

- To overcome the infinite loop in above two cases, we can use **break** as shown below.

```
i = 1;
for (;;)
{
        if (i == 6)
           break;
        printf(" %d", i);
        i++;
}
```

Here, the keyword **break** is used to terminate the loop. Once the value of i reaches 6, the for loop is terminated. We study about break in further sections.

- The semicolon at the end of for statement indicates that the body of the loop is empty.

```
for (i = 1; i <= 5; i ++);
{
        printf("%d", i);
}
```

The output of this statement is only 6. The reason is, the semicolon at the end of **for** statement indicates that no statements under **for**. The **for** loop is executed 5 times without entering into the **printf()** statement. Once the value of 'i' reaches 6, then control is transferred to the printf( ) statement which prints the value 6.

- More than one variable can be initialized within **for** statement and also more than one updation statements can be used in for statement as shown below.

```
for (i = 1, j = 10; i <= 10; i++, j++)
{
      printf("\n %d %d", i, j);
}
```

The comma operator is used to separate the two initialization statements and also two updation statements. In the above code the variables i and j are initialized to 1 and 10 respectively. The two updation statements are i++ and j++. Once the body of the loop is executed, the expressions i++ and j++ are evaluated from left to right and then the condition is tested. Since the value of the variable i is 2, the condition is true. Hence the body of the loop is executed again by printing the values of i and j respectively. This process continues as long as i <= 10 is true.

Program 21 **Write a program to find sum of natural numbers up to given number using for loop.**

```
#include <stdio.h>
void main( )
{
      int num, sum, i;
      printf("\n Enter the number of terms:");
      scanf("%d", &num);
      sum = 0;
      for (i = 1; i<=n; i++)
      {
         sum = sum+i;
      }
      printf("\n Sum of series = %d", sum);
}
```

Output:

```
Enter the number of terms :10
Sum of series = 55
```

Program 22 Write a program to find factorial of a given number using for loop.

```
#include <stdio.h>
void main()
{
      int num, fact = 1, i;
      printf ("\n Enter a number :");
      scanf ("%d", &num);
      for (i = 1; i<=num; i++)
      {
         fact = fact*i;
      }
      printf("\n Factorial of %d is %d", num, fact);
}
```

Output:

```
Enter a number : 5
Factorial of 5 is 120
```

Program 23 Write a program to calculate sum and average of five subjects for five students.

```
#include <stdio.h>
void main()
{
      int m1, m2, m3, m4, m5, sum = 0, i;
      float avg;
      for (i = 1, i <= 5; i++)
      {
         printf("\n Enter the marks of five subjects of student [%d] : \n", i);
         scanf("%d %d %d %d %d", &m1, &m2, &m3, &m4, &m5);
         sum = m1+m2+m3+m4+m5;
         avg = sum/5;
         printf("\n Total marks of student [%d] : %d", i, sum);
         printf("\n Average marks of student [%d] : %2.2f", i, avg);
      }
}
```

Output:

```
Enter the marks of five subjects of student [1] :
65 68 72 86 54
Total marks of student [1] : 345
Average marks of student [1] : 69.00
```

Program 24 Write a program to check whether a given number is prime or not.

Prime number is one which is divisible by 1 and by itself. Numbers 2, 3, 5, 7, 11 ... are prime numbers. To test the prime property of a given number, the number must be divided by all the integers from 2 to half of the number itself. (or it can be divided from 2 to square root of the number itself). If remainder becomes zero, then number is not a prime. If all the numbers between 2 and half the number does not divide the number equally, then the number is prime number.

```
#include <stdio.h>
void main()
{
      int num, i, flag = 0;
      printf("\n Enter a number :");
      scanf("%d", &num);
      for (i = 2; i <= num/2; i++)
```

```
    {
        if (num %i == 0)
        {
            flag = 1;
            break;
        }
    }
    if (flag == 1)
        printf("\n The number is not prime");
    else
        printf("\n The number is prime");
}
```

Output:

```
Enter a number : 11
The number is prime
```

7.5 Jumps in Loops

While executing any loop, it becomes necessary to skip a part of the loop or to leave the loop as soon as certain condition becomes true, that is called jumping out of loop. C language allows jumping from one statement to another within a loop as well as jumping out of the loop.

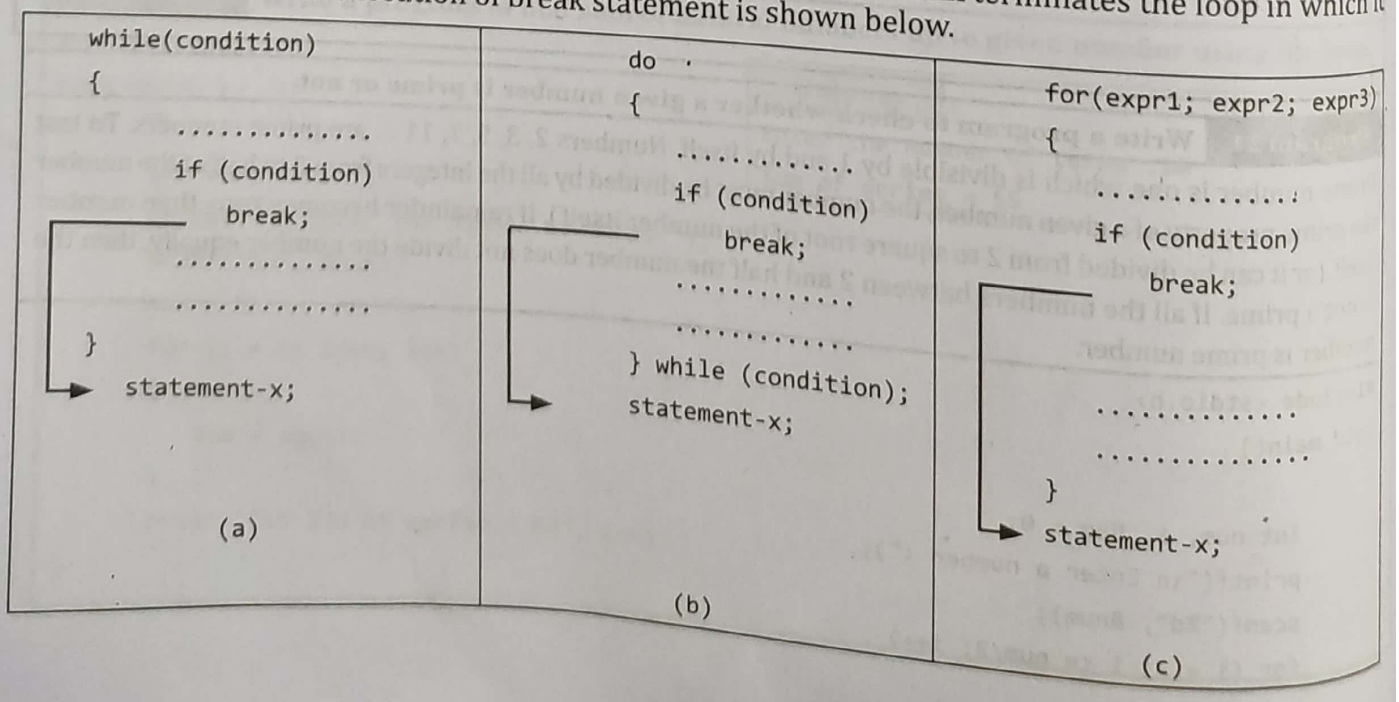
We may come across a situation where we need to terminate the execution of a loop (or) skip part of a loop. **The break (or) goto statement** can be used to terminate the execution of a loop (jumping out of loop) and **the continue statement** can be used to skip part of a loop.

7.5.1 The break keyword

We have studied the keyword **break** in a **switch** statement. Where break is used to terminate the **switch** construct. Looping constructs discussed so far executes a body of the loop repeatedly as long as the condition is TRUE. Execution of a looping construct is terminated only when the condition is evaluated to FALSE.

But in some situations, we may require to terminate the loop based on specified condition before to the condition is evaluated to FALSE.

The **break** keyword is used to terminate any of the looping constructs. This keyword is always used in connection with **if** within a looping construct. The keyword **break** terminates the loop in which it is present. The flow of execution of break statement is shown below.



Example 1

Consider the following code.

```
i = 1;
while (i <= 5)
{
    if (i == 3)
        break;
    printf("%d", i);
    i++;
}
```

The above code prints the output as 1 2. Once the value of 'i' become 3, then it checks the condition in **if**, the condition (i == 3) becomes true and break is executed which terminates the execution of the loop. Hence it has printed only 1 and 2.

Example 2

Consider the following code.

```
i = 1;
for (; ;)
{
    printf("%d", i);
    i++;
    if (i==6)
        break;
}
```

The for (; ;) statement is called infinite loop since the condition is omitted. To terminate the loop, we must use **break** keyword. The variable 'i' starts with one, and incremented by 1 each time. When i becomes 6, the condition of **if** is satisfied, and thereby the loop is terminated.

7.5.2 The Continue Keyword

In some situations, we may require to skip a set of statements from particular point in the body of the loop upto the end and control is to be taken back to the beginning of loop for the next iteration. In such situations, we can use **continue** keyword.

When the **continue** is encountered, it transfers the control back to the loop condition by skipping the rest of the statements of the body of the loop. The **continue** works only with loops. **The keyword continue cannot be used in switch construct**. The transfer of control of **continue** statement in the looping constructs is shown below.

```
while(condition)
{
    ..............
    if (condition)
        continue;
    ..............
    ..............
}
statement-x;
```

(a)

```
do
{
    ..............
    if (condition)
        continue;
    ..............
    ..............
} while (condition);
statement-x;
```

(b)

```
for(expr1; expr2; expr3)
{
    ..............
    if (condition)
        continue;
    ..............
    ..............
}
statement-x;
```

(c)

Example 1

Consider the following code.

```
for (i = 1; i < = 5; i++)
{
        if (i == 2)
            continue;
        printf("%d", i);
}
```

The above code generates the output as

1 345.

When the value of 'i' becomes 2, the condition (i == 2) is satisfied and **continue** statement is executed and control goes to next iteration of the loop. After which 'i' becomes 3 and (3 <= 5) is true and 3 is printed in the body of the loop. Similarly it prints 4 and 5. When the value of 'i' becomes 6, then the loop is terminated.

Example 2

Consider the following code

```
sum = 0
for(i = 1; i <= 100; i ++)
{
        if (i % 2 == 0)
            continue;
        sum = sum+i;
}
printf("Sum of numbers = %d", sum);
```

In the above code, the **for** loop determines sum of numbers upto 100, skipping all numbers **divisible by 2.** Whenever the value of **i** is divisible by **2,** the statement **sum = sum + i** is skipped and control is sent back to the beginning of the loop for the next iteration. So, finally the above code is used to sum all odd numbers from 1 to 100.

What are the jumping statements and how these work?

Jumping statements are used to transfer program control to one location to other location.

Example: goto, break, continue

What is infinite loop?

A loop which is never finished is known as infinite loop, it means the looping condition is always true, so that loop never terminate.

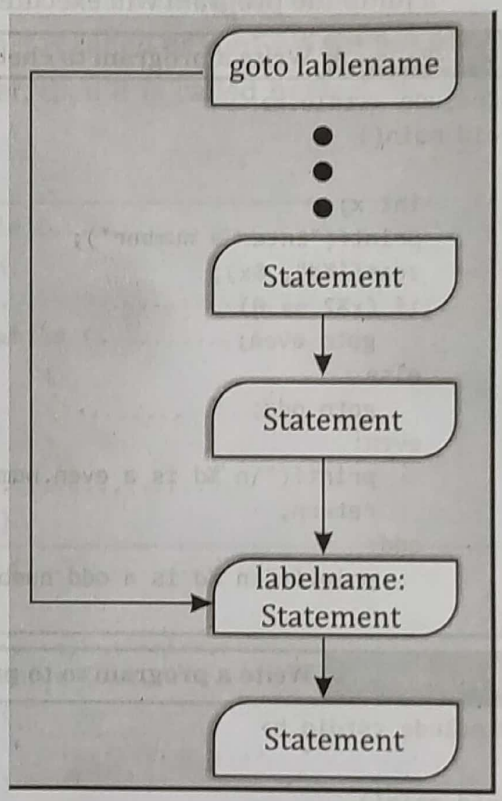
Example: while(1) { printf("Hello"); }

Can we use continue statement without using loop?

No, continue statement can only be used within the loops only, it can be any loop while, do while or for. If we use continue statement without using loop , there will be a compiler error "misplaced continue".

7.5.3 Goto Statement and Labels

The goto statement is known as **jump statement** in C. As the name suggests, goto is used to transfer the program control to a predefined label. The goto statment can be used to repeat some part of the code for a particular condition. It can also be used to break the multiple loops which can't be done by using a single break statement. However, using goto is avoided these days since it makes the program less readable and complicated. In practice it is always easy to write code without goto statement.



The goto statement allows us to transfer control of the program to the specified label. The goto statement moves the execution of the program to another statement. The transfer of the control is unconditional and one-way.

A **label** is a name that is formed with the same rules as variable names, and must be immediately, followed by a colon. The label is placed directly before the statement to which the branch is to be made, and must appear in the same function as the goto.

The label is an identifier. When the goto statement is encountered, the control of the program jumps to label: and starts executing the code.

Syntax	Example

`goto labelname;`
 	
`labelname: statements to be executed`
	`goto start;`
	
	`start: printf ("This is labeled statement");`

The above example will cause the program to branch immediately to the statement that is preceded by the label **start**. This label can be located anywhere in the function, before or after the goto statement.

The use of goto statement is unstructured programming. Moreover, use of goto statement leads to inefficient code being generated by the compiler.

The rules for label :

- Must be defined within a function.
- Each label in one function must have a unique name. It cannot be a reserved C word.

- C has a separate namespaces for identifiers and labels, so we can use the same name for a variable and a label.
- Must be followed by a statement. It is called as "labeled statement".
- Labels has function scope. Therefore the label must have a unique name within that function and is not accessible outside the function, where it was defined.

Note : When the control reaches the goto statement, the control of the program "jumps" to the label. Then the execution continues normally. If the execution reaches the labeled statement without a jump the program will execute it just like any other.

Program 25 **Write a program to check whether a number is an even or odd using goto statement.**

```
#include <stdio.h>
void main()
{
      int x;
      printf("Enter a number");
      scanf(%d", &x);
      if (x%2 == 0)
         goto even;
      else
         goto odd;
      even:
         printf("\n %d is a even number");
         return;
      odd:
         printf("\n %d is a odd number");
}
```

Output:

```
Enter a number : 5
5 is a odd number
```

Program 26 **Write a program to to print multiplication table of N using goto statement.**

```
#include <stdio.h>

void main()
{
      int count,n,result;

      printf("Enter number: ");
      scanf("%d",&n);

      count=1;

      start:
      if(count<=10)
      {
         result=n*count;
         printf("%d * %d = %d \n",n,count,result);
         count++;
         goto start;
      }
}
```

Output:

```
Enter number: 5
5 * 1 = 5
5 * 2 = 10
5 * 3 = 15
5 * 4 = 20
5 * 5 = 25
5 * 6 = 30
5 * 7 = 35
5 * 8 = 40
5 * 9 = 45
5 * 10 = 50
```

7.6 Nested Loops

A **loop** construct enclosed within another loop construct is known as **nesting of loops.** The nesting of loops is necessary when a set of statements are to be executed as long as two different conditions are satisfied.

When creating a loop inside another, indenting should be properly made to the nested loops so that user can read easily to determine which statements are contained in which loop. **Any looping construct can nest with any of the looping constructs.**

- If the **for** loops are nested one inside the other, then it is called **nested-for loops.**
- If the while loops are nested one inside the other, then it is called **nested-while loops.**
- If the do.. while loops are nested one inside the other, then it is called **nested do.. while loops.**

The different forms of nested loops is shown below.

(a)
```
for (.................)
{
   ......................
   for (...........)
   {
      ...................
   }
   ........................
}
```

(b)
```
while (.................)
   {
   ........................
   while (...........)
      {
         ...................
      }
   ..................
   }
```

(c)
```
do
{
   ......................
   do
   {
      ..................
   } while(.........);
   ......................
} while(.........);
```

(d)
```
for (.................)
{
   ......................
   while (...........)
   {
      ................
   }
   ...................
}
```

(e)
```
for (...................)
{
   ................
   for (...................)
   {
      ......................
      while (............)
      {
         ..................
      }
      ......................
   }
   ........................
}
```

(f)
```
do
   {
   ......................
   for (.................)
         {
            ...................
         for (.................)
            {
            ...................
            }
         ........................
         }
   ....................
   } while (..........);
```

- Like above forms, we can nest any loop with any of the other constructs. There is no restriction regarding number of loops that can be nested. All same type of loops or different types of loops may participate in the nesting.
- In case a **break** is used within the inner loop of two nested loops, execution of **break** terminates execution of inner loop and the control is sent to the outer loop.

```
for (.................)                    /* outerloop*/
{
    ................
    while (.............)                   /* innerloop*/
    {
        .............
        if (condition)
            break;

        ..............
        ..............
    }
    ..............
    ..............
}
```

- **Continue** statement used in a particular loop of nested loops, will send the control back to the beginning of that particular loop.

```
for   (.............)
{
    .............
    while (..........)
    {
        .............
        if (condition)
            continue;

        ..............
        ..............
    }
    ..............
    ..............
}
```

Example 1

Consider the following code.

```
int i, j;
for (i = 1; i <= 3; i++)
{
   for (j = 1; j<=5; j++)
   {
      printf("\n i = %d j=%d", i, j);
   }
}
```

The above program has nesting of two for loops. The outer loop variable 'i' is initialized with value 1, the condition (i < = 3) is tested and since it is true the control shifts to inner loop The inner loop variable 'j' is initialized to 1 and (j < = 5) becomes true, then body of the inner loop is executed by printing the value of i and j as **1** and **1**. The body of inner loop is executed **5** times for i = 1 and again inner loop is executed 5 times for i = 2 and again inner loop is executed 5 times for i = 3. The sample output is.

i = 1 j = 1	i = 2 j = 1	i = 3 j = 1
i = 1 j = 2	i = 2 j = 2	i = 3 j = 2
i = 1 j = 3	i = 2 j = 3	i = 3 j = 3
i = 1 j = 4	i = 2 j = 4	i = 3 j = 4
i = 1 j = 5	i = 2 j = 5	i = 3 j = 5

Program 27 **Write a program to print the multiplication table of 1 to 10.**

```
#include <stdio.h>
void main()
{
   int i, j;
   for (i = 1; i <= 10; i ++)
   {
      for (j = 1; j <= 10; j ++)
      {
         printf("\n %d * %d = %d", i, j, i*j);
      }
   }
}
```

Explanation

In the above nested loop, for each value of i, the inner loop is executed 10 times. When i = 1, the inner loop executes 10 times after which inner loop is terminated and control is sent back to the outer loop. In outer loop the value 'i' becomes 2 and again inner loop is executed 10 times. Total number of iterations is 10 × 10 = 100. The outer loop is terminated when the value of 'i' becomes more than 10.

Program 28 Write a program to perform addition of three loop variables using nested for loops.

```
#include <stdio.h>
main()
{
        int i, j, k;
        for (i = 1; i <= 2; i++)            /* outer loop*/
        {
            for (j = 1; j <= 2; j++)        /* middle loop*/
            {
                for (k = 1; k <= 2; k++)    /* inner loop*/
                {
                    printf("\n i = %d j = %d k = %d sum = %d", i, j, k, (i+j+k));
                }
            }
        }
}
```

Output:

```
i = 1  j = 1  k = 1  sum = 3
i = 1  j = 1  k = 2  sum = 4
i = 1  j = 2  k = 1  sum = 4
i = 1  j = 2  k = 2  sum = 5
i = 2  j = 1  k = 1  sum = 4
i = 2  j = 1  k = 2  sum = 5
i = 2  j = 2  k = 1  sum = 5
i = 2  j = 2  k = 2  sum = 6
```

Explanation

For each value of i, middle loop executed 2 times and inner loop is executed 2 times. The outer loop executes two times. Therefore total number of iterations = 2 × 2 × 2 = 8. The following table shows the execution of the above program.

Outer loop variable	middle loop variable	inner loop variable	sum
i = 1	j = 1	k = 1	sum = 3
		k = 2	sum = 4
	j = 2	k = 1	sum = 4
		k = 2	sum = 5
i = 2	j = 1	k = 1	sum = 4
		k = 2	sum = 5
	j = 2	k = 1	sum = 5
		k = 2	sum = 6

When i = 1, and j = 1, the inner loop executes 2 times. Similarly when i=1 and j = 2, the inner loop executes 2 times. The outer loop is terminated when i becomes 3.

Program 29 Write a program to display the stars as shown below.

```
*
*   *
*   *   *
*   *   *   *
*   *   *   *   *
```

```
#include <stdio.h>
main()
{
    int i, j, n;
    printf("\n Enter the number of lines :");
    scanf("%d", &n);
    for (i = 1; i <= n; i++)
    {
       for (j = 1; j <= i; j++)
       {
          printf(" * ");
       }
       printf("\n");
    }
}
```

Output:

```
Enter the number of lines : 6
*
*     *
*     *     *
*     *     *     *
*     *     *     *     *
*     *     *     *     *     *
```

Explanation

The inner loop is used to print the start (*). To display them in different lines second printf( ) statement is used. When i = 1, the 'j' value ranges from 1 to 1 and hence it is printed only one star on first line. When i = 1, the 'j' ranges from 1 to 2 and hence it is printed two stars on second line. This process is continued until 'i' is less than (or) equal to '**n**'.

Program 30 **Write a program to print the stars as shown below.**

```
* * * * *
* * * *
* * *
* *
*
```

```
#include <stdio.h>
void main()
{
    int i, j, n;
    printf("\n Enter the number of lines :");
    scanf("%d", &n);
    for (i = n; i > 0; i--)
    {
       for (j = 1; j <= i; j++)
          printf(" * ");
       printf("\n");
    }
}
```

Output:

```
Enter the number of lines : 5
 *  *  *  *  *
 *  *  *  *
 *  *  *
 *  *
 *
```