1. CONTROL STATEMENTS

The default order of execution in a C program is top-down. Execution starts at the beginning of the *main()* function and progresses, statement by statement, in the order in which they are written, until the end of *main()* is reached. This is called *sequential execution*. During sequential execution, each instruction will be executed only once. Programs of this type are unrealistically simple and did not include any statements that tests to determine if certain conditions are true or false, statements that are repeatedly executed and statements that execute on a selective basis. Most C programs that are of practical interest make extensive use of features such as these.

For example, a realistic C program may require that a logical test be carried out at some particular point within the program. One of several possible actions will then be carried out, depending on the outcome of the logical test. This is known as *branching*. There is also a special kind of branching, called *selection*, in which one group of statements is selected from several available groups. In addition, the program may require that a group of instructions be executed repeatedly, until some logical condition has been satisfied. This is known as *looping*. Sometimes the required number of repetitions is known in advance; and sometimes the computation continues indefinitely until the logical condition becomes true. All of these operations can be carried out using the various *control statements* included in C.

Control statements are used to create special program features, such as logical test, loops and branches. There are three types of control statements in C. They are,

1. Branching and selection statements or Conditional statements, such as *if* and *switch*.
2. Iteration statements or Loop statements, such as *while*, *for* and *do-while*.
3. Jump statements such as *break*, *continue*, *goto* and return.

2. SELECTION STATEMENTS

A selection statement selects a group of statements from several available groups, depending on the outcome of a logical test. C supports two selection statements: if and switch.

2.1. THE *if* STATEMENT

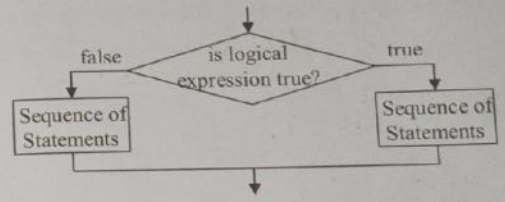
The *if* statement is used to carry out a logical test and then take one of two possible actions, depending on the outcome of the test (i.e., whether the outcome is true or false). The general form of the *if* statement is

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


where the *statement1* or *statement2* can be either simple or compound statement.

The else clause is optional. The expression must be placed in parentheses.

If the expression has a nonzero value (i.e., if the *expression* is true) then *statement1* will be executed. Otherwise (i.e., if the *expression* is *false*), *statement2* will be executed. This form of *if* statement is represented in the following flowchart.

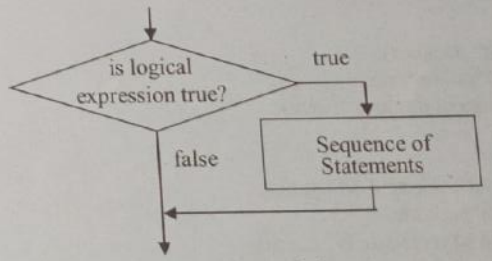


The *else* portion of the *if* is optional. Thus in its simplest form, the *if* statement can be written as

```
if (expression) statement;
```

In this form, the *statement* will be executed only if the *expression* has a nonzero value (i.e., if the expression is *true*). If the expression has a value of zero (i.e., if the expression is false), then the statement will be ignored.

This form of *if* statement is represented in the following flowchart



Several representative *if* statements are shown below.

1. `if(x<0) printf("%f",x);`
2.
```
if(p>0)
    credit = 0;
```
3.
```
if(x<=3.0)
    { y=3*pow(x,2);
```

selection (S)	Iteration (S)	Jump (S)
if switch	while do while for	break continue goto

```
    printf("%f\n",y);
  }
4. if((balance<1000.0)||(status=='R'))
    printf("%f",balance);

5. if((a>=0)&&(b<=5)
   { xmid = (a+b)/2;
     Ymid = sqrt(xmid);
   }

6. if(status=='S')
    tax=0.02*pay;
   else
    tax=0.14*pay;

7. if(p>0)
   { printf("Overdue");
     credit = 0;
   }
   else
     credit=1000.0;

8. if(x<=3.0)
    y=3*pow(x,2);
   else
    y=2*pow((x-3),2);
   printf("%f\n",y);

9. if (circle)
   {  scanf("%f",&radius);
      area=3.14*radius*radius;
      printf(" Area of circle is ", area);
   }
   else
   { scanf("%f, %f", &length,&breadth);
     area=length*breadth;
     printf("Area of rectangle is " ,area);
   }
```

It is possible to nest (i.e., embed) *if-else* statements, one within another. The *if* statements can be nested in several different forms. The most general form of two-layer nesting is

```
if(e1)
  if (e2)
    s1;
  else
    s2;
else
  if (e3)
    s3;
  else
    s4;
```


where $e1$,$e2$ and $e3$ represent logical expressions and $s1$,$s2$,$s3$ and $s4$ represent statements. One complete *if-else* statement will be executed if $e1$ is nonzero (*true*), and another complete *if-else* statement will be executed if $e1$ is zero (*false*).

Some other forms of two-layer nesting are

```
if(e1)          if(e1)          if(e1)          if(e1)
  s1;             s1;             if(e2)          if(e2)
else            else                s1;             s1;
  if(e2)          if(e2)          else            else
    s2;             s2;             s2;             s2;
                  else            else
                    s3;             s3;
```

While evaluating the nested *if* statement the rule is that the *else* clause is always associated with closest preceding unmatched (i.e., else-less) *if*.

In some situations it may be desirable to nest multiple *if-else* statements as an *else-if* ladder, in order to create a situation in which one of several different courses of action will be selected. Its general form is

```
if(e1)
  s1;
else if(e2)
    s2;
  else if(e3)
      s3;
    else if
      .
      .
      .
    else
      sn;
```

The conditions are evaluated from top to downward. As soon as a true condition is found, the statement associated with it is executed and the rest of the ladder is bypassed. If none of the conditions are true, the final *else* is executed. That is, if all other conditional test fail, the last *else* statement is performed. If the final *else* is not present, no action takes place if all other conditions are *false*.

A simple if statement of the form

if(expression)

```
    statement1;
else
    statement2;
```

can be replaced by a conditional expression that uses the conditional operator (?:), as shown below:

```
expression1 ? expression2 : expression3
```

The *expression1* is interpreted as a Boolean expression and its value which of the two expressions, *expression2* or *expression3*, is to be evaluated.

The *expression1* is evaluated first. If *expression1* is true (i.e., if its value is non zero), then *expression2* is evaluated and this becomes the value of the conditional expression. However, if *expression1* is false (i.e., if its value is zero), the *expression3* is evaluated and this becomes the value of the conditional expression. Just like if statement, only one of the embedded expressions (either *expression2* or *expression3*) is evaluated when determining the value of a conditional expression.

For example, the *if* statement

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

can be expressed as a conditional expression

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

2.2. THE *switch* STATEMENT

switch is a multiple-branch selection control statement in which one group of statement is selected from several available groups. The selection is based upon the current value of an *expression*, which is included within the switch statement. The general form of the *switch* statement is

```
switch(expression) statement;
```

where expression results in an integer value. The expression can also be of type char, since individual characters have equivalent integer values.

The embedded statement is generally a compound statement that specifies alternate courses of action. Each alternative is expressed as a group of one or more individual statements within the overall embedded statement. For each alternative, the first statement within the group must be preceded by one or more case labels. The case labels identify the different groups of statements and distinguish them from one another. The case labels must therefore be unique within a given switch statement. The general form of each group of statement is


```
case expression :
statement1;
statement2;
...
statementn;
```

or, when multiple case labels are required,

```
case expression1 :
case expression2 :
...
case expressionm :
statement1;
statement2;
...
statementn;
```

where *expression1*, *expression2*, ..., *expressionm* represent constant, integer-valued expressions. Usually, each of these expressions will be written as either as an integer constant or a character constant. Each individual statement following the case label may be either simple or compound.

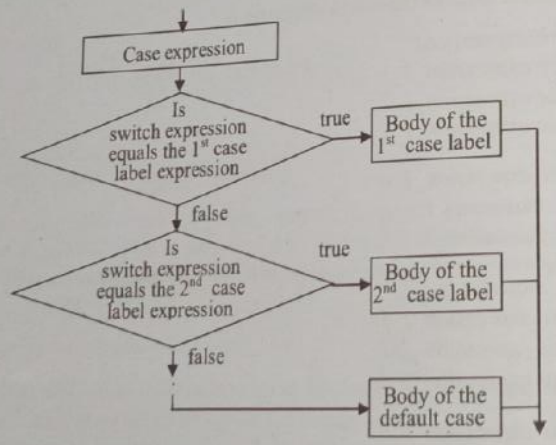
Thus, the general form of the *switch* statement is

```
switch(expression)
{ case expression_1_1:
  case expression_1_2:
  case expression_1_3:
  ...
  case expression_1_n:
      statement_1;
      statement_2;
      ...
      statementn
  case expression_2_1:
  case expression_2_2:
  case expression_2_3:
  ...
  case expression_2_n:
      statement_1;
      statement_2;
      ...
      statement_n
  ...
  case expression_m_1:
  case expression_m_2:
  case expression_m_3:
```

```
...
case expression_m_n:
    statement_1;
    statement_2;
    ...
    statement_n
default:
    statement_1;
    statement_2;
    ...
    statement_n
}
```

When the *switch* statement is executed, the *expression* in the *switch* statement is evaluated and the control is transferred directly to the group of statements whose case label value matches the value of the *expression*. If none of the case-label values matches the value of the *expression*, then none of the groups within the *switch* statement will be selected and the control is transferred directly to the statement that follows the *switch* statement.

The following flowchart illustrates the operation of the *switch* statement



One of the labelled groups of statements within the *switch* statement may be labelled *default*. This group will be selected if none of the case labels matches the value of the *expression*. The *default* group may appear anywhere within the *switch* statement – it need not necessarily be placed at the end. If none of the case labels matches the value of the *expression* and the *default* group is not present, then no action will be taken by the *switch* statement.

The following program illustrates the use of *switch* statement.


```
/* To input the first letter of the colours in VIBGYOR and to display the name of the
colour */
#include<stdio.h>
main()
{   char c;
    printf("Enter any letter from VIBGYOR : ");
    switch(c=getchar())
    {   case 'v':
        case 'V': printf("Violet");
                  break;
        case 'i':
        case 'I': printf("Indigo");
                  break;
        case 'b':
        case 'B': printf("Blue");
                  break;
        case 'g':
        case 'G': printf("Green");
                  break;
        case 'y':
        case 'Y': printf("Yellow");
                  break;
        case 'o':
        case 'O': printf("Orange");
                  break;
        case 'r':
        case 'R': printf("Red");
                  break;
        default: printf("Bad data");
    }
}
```

The switch statement in the above program can be written more concisely as

```
switch(toupper(c=getchar()))
{   case 'V': printf("Violet");
              break;
    case 'I': printf("Indigo");
              break;
    case 'B': printf("Blue");
              break;
    case 'G': printf("Green");
              break;
    case 'Y': printf("Yellow");
              break;
    case 'O': printf("Orange");
              break;
```

```
    case 'R': printf("Red");
              break;
    default: printf("Bad data");
}
```

Here multiple case labels for each group of statements are avoided by the use of the library function toupper.

The *break* statement is used to exit from the *switch*. The break statement causes a transfer of control out of the entire switch statement, to the first statement following the *switch* statement.

The *switch* statement can be nested by having a *switch* as a part of the statement sequence of an outer *switch*. Even if the *case expressions* of the inner and outer switch contain common values, no conflicts arise. For example, the following code fragment is perfectly acceptable.

```
switch(x)
{ case 1:
    switch(y)
    { case 0: printf("Divide by zero error\n");
              break;
  case 1:process(x,y)
              break;
    }
  case 2:
  ...
```

The *switch* statement can be considered a good alternative to the nested if-else statements that test for equality. The *switch* differs from the *if* statement in that *switch* can only test for equality, whereas *if* can evaluate any type of relational or logical expression.

3. ITERATION STATEMENTS

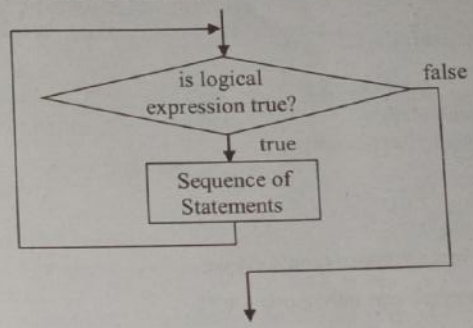
The *iteration statements* (also called *loop statements*) allow a set of instructions to be repeatedly executed until a certain condition is reached. This condition may be predetermined (as in *for* loop) or open-ended (as in *while* and *do-while* loops)

3.1. THE *while* STATEMENT

The *while* statement is used to carryout looping operations in which a group of statements is executed repeatedly as long as the given condition is *true*. The general form of the *while* statement is

while (expression) statement;

The *statement* will be executed repeatedly, as long as the *expression* is true (i.e., as long as the *expression* has a nonzero value). The *statement* can be either simple or compound. The *statement* must include some feature that eventually alters the value of the *expression*, thus providing stopping condition for the loop. The flowchart given beow illustrates the functioning of the *while* loop.



The following program illustrates the use of the while statement.

```
/* To display the whole number up to 100*/
#include<stdio.h>
main()
{   int number=0;
    while(number<=100)
    {   printf("%d ",number);
        number++;
    }
}
```

This program can be written more concisely as

```
/* To display the whole number up to 100*/
#include<stdio.h>
main()
{   int number=1;
    while(number<=100)
    printf("%d ",number++);
}
```

The *while* statement is particularly useful when the number of passes through the loop is not known in advance. In such situations, the looping action needs to be continued indefinitely, as long as the specified logical condition is satisfied.

The following example illustrates the use of *while* statement in a situation where the exact number of passes through the loop is unknown.

```
/* To convert a lowercase text into uppercase */
#include<stdio.h>
```

```
main()
{ int count=0;
    char text[80];
    text[count]=getchar();
    while(text[count]!= '\n')
    { count++;
    text[count]=getchar();
    }
    count = 0;
    while(text[count]!='\n')
    {   putchar(toupper(text[count]));
    count++;
    }
}
```

This program can be written more concisely as

```
/* To convert a lowercase text into uppercase */
#include<stdio.h>
main()
{   int count=0;
    char text[80];
    while((text[count++]=getchar()) != '\n');
    count = 0;
    while(text[count]!='\n')
    putchar(toupper(text[count++]));
}
```

The *while* is an *entry-controlled* loop statement. In *while* statement, the test for continuation of the loop is carried out at the beginning of each pass. Therefore, the statement part will not be executed even once if condition is initially false. The *while* statement is well suited if the number of passes through the loop is not known in advance and the looping action is to be continued indefinitely until the specified logical condition is satisfied.

If the number of iterations is not known in advance, the while loop must use some mechanism for exiting from the loop. For example, consider a program that reads marks of an arbitrary number of students for determining the class average. How can the program determine when to stop the input of grades? How will it know when to calculate and print the class average?

One way to solve this problem is to use a special value called a *sentinel value* (also called a *signal value*, a *dummy value*, or a *flag value*) to indicate "end of iteration." The user types in grades until all legitimate grades have been entered. The user then types the sentinel value to indicate that the last grade has been entered. Sentinel-controlled repetition is often called indefinite repetition because the number of repetitions is not known before the loop begins executing.

Clearly, the sentinel value must be chosen so that it cannot be confused with an acceptable input value. Since grades on a quiz are normally nonnegative integers, –1 is an acceptable sentinel value for this problem. Thus, a run of the class average program might process a stream of inputs such as 95, 96, 75, 74, 89 and –1. The program would then compute and print the class average for the grades 95, 96, 75, 74, and 89 (–1 is the sentinel value, so it should not enter into the averaging calculation).

The following program illustrates how the sentinel value can be used for loop controlling loop iterations.

```
/* The sentinel controlled loop*/
#include<stdio.h>
main()
{   int i=0, mark=0, sum=0;
    printf("Enter the marks of course for each student:\n");
    while(mark>=0)
    {     printf("\nEnter the Marks (A Negative Mark to End Data Entry) of Student - %d
                    : ",i++);
      scanf("%d",&mark);
      if(mark>=0) sum+=mark;
    }
      printf("Average Mark of the Class for the Course is: %f", (float)sum/(i-1));
}
```

3.2. THE *do-while* STATEMENT

The *do-while* statement is also used to carry out looping operations, in which a group of statement is executed repeatedly, as long as the given condition is *true*. The general form of while statement is

do statement while (expression);

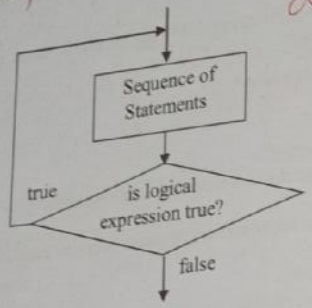
The *statement* will be executed repeatedly, as long as the value of the expression is *true* (i.e., nonzero). The *statement* can be either simple or compound. The *statement* must include some feature that eventually alters the value of the expression, thus providing stopping condition for the loop.

In *do-while* statement, the test for continuation of the loop is carried out at the end of the each pass. Therefore, the statement part will be always executed at least once even if the condition is initially false. The do-while statement is well suited if the number of passes through the loop must be at least one and the exact number passes through the loop is not known in advance. The flowchart given alongside illustrates the functioning of the while loop.

The following example illustrates the use of do-while loop.

```
/* display the integers 0 through 100 */
```

```
#include<stdio.h>
main()
{   int digit = 0;
    do
        printf("%d ",digit++);
    while(digit<=100);
}
```



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

The following example illustrates that the use of *do-while* loop is preferred over the while loop if the number passes through the loop is at least one.

/* To display the sum and average of a set of numbers. The program should terminate only when a zero is entered */

```
#include<stdio.h>
main()
{  int count=0;
   float number,sum=0.0,average;
   do
   {   printf("Enter your No-%d (Zero to Stop) : " ,++count);
   scanf("%f",&number);
   sum = sum + number;
   }
   while(number != 0);
   average = sum/(count-1);
   printf("Sum is %g\nAverage is %g",sum,average);
}
```

3.3. THE *for* STATEMENT

The *for* statement is also an entry-controlled loop used to carry out looping

operations, in which a group of statement is executed repeatedly for a specific number of times. This statement includes an expression that specifies an initial value for an index, another expression that determines whether or not the loop is continued, and a third expression that allows the index to be modified at the end of each pass.

The general form of *for* statement is

for(expression1;expression2;expression3) statement;

The *for* loop allows many variations, but its most common form work like this: The *expression1* is used to initialise some parameter (called *index*) that controls the looping action, *expression2* represents a logical condition that must be true for the loop to continue execution, and *expression3* is used to alter the value of the index initially assigned by *expression1*. Typically, *expression1* is assignment expression, *expression2* is a logical expression and *expression3* is a unary expression or an assignment expression.

When the *for* statement is executed, *expression2* is evaluated and tested at the beginning of each pass through the loop, and *expression3* is evaluated at the end of each pass. Thus, the *for* statement is equivalent to

```
expression1;
while(expression2)
{   statement;
    expression3;
}
```

The looping action will continue as long as the value of the *expression2* is not zero, i.e., as long as the logical condition represented by *expression2* is *true*. The operation of the *for* loop is illustrated in the flowchart given below.

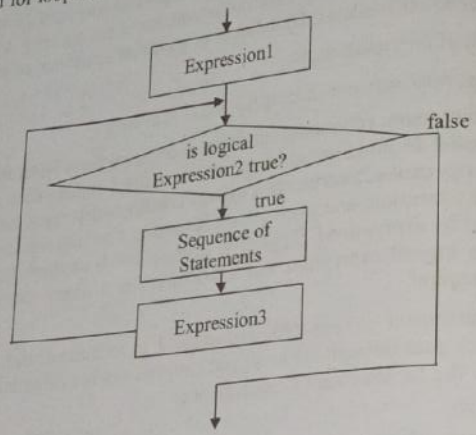
The *for* statement, like the *while* and *do-while* statements, can be used to carry out looping actions where the number passes through the loop is not known in advance. Because of the features that are built into the *for* statement, however, it is particularly well suited for loops in which the number of passes is known in advance.

The following example illustrates the use of *for* statement.

```
/* To display the integers 0 through 100 */
#include<stdio.h>
main()
{   int digit;
    for(digit=0;digit<=100;++digit)
      printf("%d ",digit);
}
```

The *for* statement, like the *while* and *do-while* statements, can be used to carry out looping actions where the number passes through the loop is not known in advance. Because of the features that are built into the *for* statement, however, it is particularly

well suited for loops in which the number of passes is known in advance.



The following example illustrates the use of *for* statement.

```
/* To display the integers 0 through 100 */
#include<stdio.h>
main()
{   int digit;
    for(digit=0;digit<=100;++digit)
      printf("%d ",digit);
}
```

The *expression1* and *expression3* can be omitted from the syntax of the *for* statement, if other means are provided for initialising the index and/or altering the index. This is illustrated in the following example.

```
/*To display the integers 0 through 100 */
#include<stdio.h>
main()
{   int digit=0;
    for(;digit<=100;)
        printf("%d",digit++);
}
```

However, the semicolon must be present. If *expression2* is omitted, however, it will be assumed to have a permanent value of 1 (*true*) and therefore the loop will continue indefinitely unless it is terminated by some other means, such as a *break* or *return* statement. This is illustrated in the following example.


```
/* To display the sum and average of a set of numbers.
The program should terminate only when a zero is entered */
#include<stdio.h>
main()
{ int count=0;
  float number,sum=0.0,average;
  for(;;)
  { printf("Enter your No-%d (Zero to Stop) : " , ++count);
      scanf("%f",&number);
      if (number == 0) break;
      sum = sum + number;
  }
  average = sum/(count-1);
  printf("Sum is %g\nAverage is %g",sum,average);
}
```

The use of comma operator (,) within *for* statements permits multiple expressions to appear in situations where only one expression would ordinarily be used. For example, it is possible to write

for(expression1a, expression1b; expression2; expression3a, expression3b)

where *expression1a* and *expression1b* are two expressions, separated by comma operator, where only one expression (*expression1*) would normally appear. Similarly, *expression3a* and *expression3b*, separated by comma operator, appear in place of the usual single expression. The use of comma operator in the *for* statement is illustrated in the following C program.

```
/* To display the series 0,100,1,99,2,98,...,99,1,100,0 */
#include<stdio.h>
main()
{ int a,b;
  for(a=0,b=100;a<=100;++a,--b)
    printf("%d\t%d\t",a,b);
}
```

4. JUMP STATEMENTS

Normally, loops perform a set of operations repeatedly until the loop control variable fails to satisfy the test condition. Sometimes it may be desirable to skip a part of the loop or to exit from the loop as soon as certain condition occurs. C permits jumps from one statement to another as well as a jump out of loop.

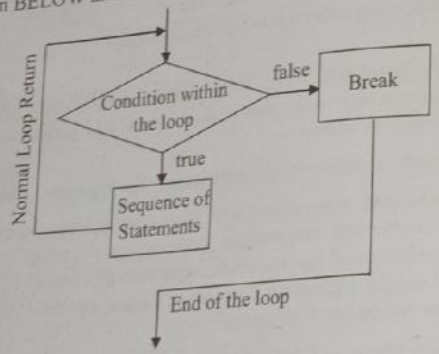
4.1. THE *break* STATEMENT

The *break* statement is used to terminate loops or to exit from a *switch*. It can be used within a *for*, *while*, *do-while* or *switch* statement. The *break* statement is simply is

written as

break;

without any embedded expressions or statements. In *switch* statement, the *break* statement causes a transfer of control out of the entire *switch* statement, to the first statement following the *switch* statement. If a *break* statement is included in *while*, *do-while* or *for* loop, the control will immediately be transferred out of the loop when the *break* statement is encountered. In the event of several nested *while*, *do-while*, *for* or *switch* statement a *break* statement will cause a transfer of control out of the immediate enclosing statement, but not out of the outer surrounding statements. The flowchart given BELOW indicates the functioning of the *break* statement.



The following program illustrates the use of *break* statement in a while loop.

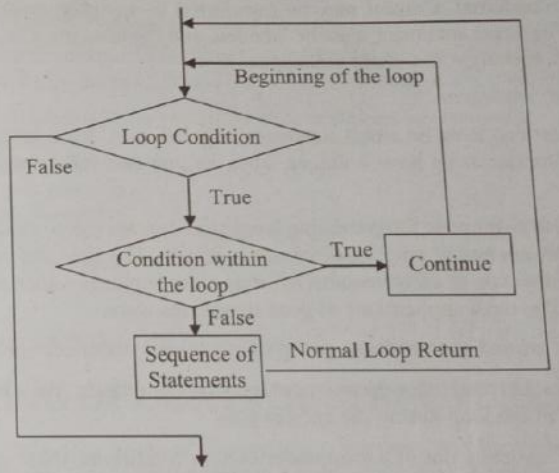
```
/* To display the sum and average of a set of numbers.
The program should terminate only when a zero is entered */
#include<stdio.h>
main()
{ int count=0;
  float number,sum=0.0,average;
  while(1)
  { printf("Enter your No-%d (Zero to Stop) : " , ++count);
    scanf("%f",&number);
    if (number == 0) break;
    sum = sum + number;
  }
 average = sum/(count-1);
 printf("Sum is %g\nAverage is %g", sum,average);
}
```


4.2. THE *continue* STATEMENT

The *continue* statement is used to bypass the remainder of the current pass through a loop. The loop does not terminate when a *continue* statement is encountered. Rather, the remaining loop statements are skipped and the computation proceeds directly to the next pass through the loop. The *continue* statement can be included within a *while*, *do-while*, or *for* statement. It is written simply as

continue;

without any embedded expressions or statements. The flowchart given BELOW illustrates the functioning of the *continue* statement.



The following program illustrates the use of *continue* statement.

/* To display the sum and average of nonnegative numbers from a set of given numbers. The program should terminate only when a zero is entered */

```
#include<stdio.h>
main()
{ int count=0;
  float number,sum=0.0,average;
  while(1)
   { printf("Enter your No-%d (Zero to Stop): ", ++count);
     scanf("%f",&number);
     if (number == 0) break;
     if (number < 0 ) continue;
     sum = sum + number;
```

```
    }
    average = sum/(count-1);
    printf("Sum is %g\nAverage is %g", sum,average);
}
```

4.3. THE goto STATEMENT

The *goto* statement is used to alter the normal sequence program execution by transferring control to some other part of the program unconditionally. In its general form, the *goto* statement is written as (used to jump from anywhere to anywhere within a C)

goto label;

where the *label* is an identifier that is used to label the target statement to which the control is transferred. Control may be transferred to anywhere within the current function. The target statement must be labelled, and the label must be followed by a colon. Thus, the target statement will appear as

label: statement;

The *statement* can even be a null statement (e.g. *label: ;*). Each labelled statement within the function must have a unique label, i.e., no two statements can have the same label.

All of the general-purpose programming languages contain a goto statement, though the modern programming practice discourages its use. The *goto* statement was used extensively, however, in early versions of some older languages, such as FORTRAN and BASIC. The most applications of *goto* statements were:

1. Branching around statements or group of statements under certain conditions.
2. Jumping to the end of a loop under certain conditions, thus bypassing the remainder of the loop during the current pass.
3. Jumping completely out of a loop under certain conditions, thus terminating the execution of a loop.

The structured features in C enable all these operations to be carried out without the use of *goto* statements. For example, branching around statements can be accomplished with the *if-else* statement; jumping to the end of a loop can be carried out with the *continue* statement; and jumping out of a loop is easily accomplished using the *break* statement. The use of these structured features is preferable to the use of the *goto* statement, because the use of *goto* tends to encourage (or at least not discourage) logic that skips all over the program where as the structured features in C requires that the entire program be written in an orderly, sequential manner. For this reason, the use of *goto* statement should generally be avoided

Occasional situation do arise, however, in which the *goto* statement can be useful. Consider, for example, a situation in which it is necessary to jump out of a doubly

nested loop if a certain condition is detected. This can be accomplished with two *if-break* statements, one within each loop, though this is awkward. This is illustrated in the following program segment.

```
/* To determine whether two array have an element on common */
flag = 0;
for(i=0;i<n;++i)
{ for(j=0,i<m,++j)
    if(x[i]==y[j])
    { flag=1;
      break;
    }
  if(flag) break;
}
if(flag) printf("Identical Elements exists");
```

A better solution in this particular situation might make use of the goto statement to transfer of both loops at once. This is illustrated in the following program segment.

```
/* To determine whether two arrays have an element on common */
flag = 0;
for(i=0;i<n;++i)
   for(j=0,i<m,++j)
     if(x[i]==y[j])
     { flag=1;
       goto found;
     }
found:
if(flag) printf("Identical Element exists");
```

5. NESTING OF CONTROL STRUCTURES

Control structures like loops and selection statements can be nested, one within another, upto any level. The different levels of control structures need not be, generated by the same type of control structure. It is essential, however, that one control structure be completely, embedded within the other - there can be no overlap.

For example, consider the use of nested *for* statement to generate the following pattern:

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

6.1 ANSWERS TO SELECT PROGRAMMING EXERCISES

1.

```
/* program to calculate the distance travelled */
#include<stdio.h>
main()
{   float distance, time, speed;
    printf("Speed in Km/Hr ? ");
    scanf("%f",&speed);
    printf("Time of Journey ? ");
    scanf("%f",&time);
    distance = speed*time;
    printf("%.2f Kms travelled within %.2f Hrs at a speed of %.2f Km/Hr.", distance, time,
                speed);
}
```

2.

```
#include<stdio.h>
main()
{ int a,b,c;
  printf("Enter Three Numbers\n");
  printf("First Number  ? ");
  scanf("%d",&a);
  printf("Second Number ? ");
  scanf("%d",&b);
  printf("Third Number  ? ");
  scanf("%d",&c);
  printf("Half of %d is %g\n",a,a/2.0);
```

```
  printf("Double of %d is %d\n",b,b+b);
  printf("Square of %d is %d\n",c,c*c);
}
```

3.

```
/* To calculate Volume of a box */
#include<stdio.h>
main()
{ float length, breadth, height,volume;
  printf("Enter Dimension of a box\n");
  printf("Length  ? ");
  scanf("%f",&length);
  printf("Breadth ? ");
  scanf("%f",&breadth);
  printf("Height  ? ");
  scanf("%f",&height);
  volume = length*breadth*height;
  printf("Volume of the box is %.3g",volume);
}
```

4.

```
/* To calculate Perimeter & Area of a rectangle */
#include<stdio.h>
main()
{ float length, breadth, perimeter, area;
  printf("Enter Dimension of a rectangle\n");
  printf("Length  ? ");
  scanf("%f",&length);
  printf("Breadth ? ");
  scanf("%f",&breadth);
  perimeter = 2*(length+breadth);
  area = length*breadth;
  printf("Perimeter of the rectangle is %.3g\n", perimeter);
  printf("Area of the rectangle is %.3g",area);
}
```

5.

```
/* To calculate circumference & area of a circle */
# define pi 3.14
#include<stdio.h>
main()
{ float radius, circumference, perimeter, area;
  printf("Enter Radius of the circle ");
  scanf("%f",&radius);
  circumference = 2*pi*radius;
```



```
  area = pi*radius*radius;
  printf("Circumference of the circle is %.3g\n", circumference);
  printf("Area of the circle is %.3g",area);
}
```

6.

```
/* to calculate surface area and volume of a sphere */
# define pi 3.14
#include<stdio.h>
main()
{ float radius, area, volume;
  printf("enter radius of the sphere ");
  scanf("%f",&radius);
  area = 4*pi*radius*radius;
  volume = (4.0/3.0)*pi*pow(radius,3);
  printf("surface area of the sphere is %.3g\n",area);
  printf("volume of the sphere is %.3g",volume);
}
```

7.

```
/* to display a number, its square, its cube and square root */
#include<stdio.h>
# include<math.h>
main()
{ float number;
  printf("enter a number: ");
  scanf("%f",&number);
  printf("the number you entered is %g\n",number);
  printf("square of %g is %g\n",number,pow(number,2));
  printf("cube of %g is %g\n",number,pow(number,3));
  printf("square root of %g is %g\n",number,sqrt(number));
}
```

8.

```
/* to convert temperature in degree celcius to fahrenheit */
#include<stdio.h>
main()
{ float celcius, fahrenheit;
  printf("enter temperature in degree celcius : ");
  scanf("%f",&celcius);
  fahrenheit = (9.0/5.0)*celcius+32;
  printf("%g degree celcius is %g fahrenheit", celcius,fahrenheit);
}
```

9.

```
/* To convert number of days into years, weeks and days */
#include<stdio.h>
main()
{ int temp,days,year,week,day;
  printf("enter the number of days : ");
  scanf("%d",&days);
  temp = days;
  year = (int)days/365;
  days = days%365;
  week = (int)days/7;
  days = days%7;
  printf("%d days equals %d year, %d week and %d days", temp,year,week, days);
}
```

10.

```
/* To read and sum up 5 amounts in rupees and round the amount to rupee */
#include<stdio.h>
#include<math.h>
main()
{ float amt1,amt2,amt3,amt4,amt5,sum,temp;
  printf("Enter Five amounts in Rupees : \n");
  printf("First Amount  ? ");
  scanf("%f",&amt1);
  printf("Second Amount  ? ");
  scanf("%f",&amt2);
  printf("Third Amount  ? ");
  scanf("%f",&amt3);
  printf("Fourth Amount  ? ");
  scanf("%f",&amt4);
  printf("Fifth  Amount  ? ");
  scanf("%f",&amt5);
  sum = amt1+amt2+amt3+amt4+amt5;
  printf("Total of %.2f, %.2f, %.2f, %.2f and %.2f is %.2f\n",
                        amt1,amt2,amt3,amt4,amt5,sum);
  temp=sum+0.5;
  printf("Rs. %.2f can be rounded to Rs. %g",sum,floor(temp));
}
```

11.

```
#include<math.h>
main()
{ float number,temp;
  printf("Enter the Number ");
  scanf("%f",&number);
```



```
  temp=number+0.5;
  printf("%g can be rounded to %g",number,floor(temp));
}
```

12.

```
/* To round a floating-point number to its nearest tenth and hundredth */
#include<stdio.h>
main()
{ float number;
  int tenth, hundredth;
  printf("Enter the Number ");
  scanf("%f",&number);
  tenth=((int)(number+5)/10)*10;
  hundredth=((int)(number+50)/100)*100;
  printf("%g can be rounded to the nearest tenth as %d\n",number,tenth);
  printf("%g can be rounded to the nearest hundredth as %d" , number,hundredth);
}
```

13.

```
/* To round an integer to its nearest tenth and hundredth */
#include<stdio.h>
main()
{ int number,tenth,hundredth;
  printf("Enter the Number ");
  scanf("%d",&number);
  tenth=((number+5)/10)*10;
  hundredth=((number+50)/100)*100;
  printf("%d can be rounded to the nearest
       tenth as %d\n",number,tenth);
  printf("%d can be rounded to the nearest
       hundredth as %d",number,hundredth);
}
```

15.

```
/* To find the denomination of an amount*/
#include<stdio.h>
main()
{ int amt,temp,n2000,n1000,n500,n200,n100;
  int n50,n20,n10,n5,n2,n1,total;
  printf("Enter the Amount ");
  scanf("%d",&amt);
  temp = amt;
  n2000 = amt/1000;
  amt = amt%2000;
  n1000 = amt/1000;
```

9.

```
/* To convert number of days into years, weeks and days */
#include<stdio.h>
main()
{ int temp,days,year,week,day;
  printf("enter the number of days : ");
  scanf("%d",&days);
  temp = days;
  year = (int)days/365;
  days = days%365;
  week = (int)days/7;
  days = days%7;
  printf("%d days equals %d year, %d week and %d days", temp,year,week, days);
}
```

10.

```
/* To read and sum up 5 amounts in rupees and round the amount to rupee */
#include<stdio.h>
#include<math.h>
main()
{ float amt1,amt2,amt3,amt4,amt5,sum,temp;
  printf("Enter Five amounts in Rupees : \n");
  printf("First Amount  ? ");
  scanf("%f",&amt1);
  printf("Second Amount  ? ");
  scanf("%f",&amt2);
  printf("Third Amount  ? ");
  scanf("%f",&amt3);
  printf("Fourth Amount  ? ");
  scanf("%f",&amt4);
  printf("Fifth  Amount  ? ");
  scanf("%f",&amt5);
  sum = amt1+amt2+amt3+amt4+amt5;
  printf("Total of %.2f, %.2f, %.2f, %.2f and %.2f is %.2f\n",
                    amt1,amt2,amt3,amt4,amt5,sum);
  temp=sum+0.5;
  printf("Rs. %.2f can be rounded to Rs. %g",sum,floor(temp));
}
```

11.

```
#include<math.h>
main()
{ float number,temp;
  printf("Enter the Number ");
  scanf("%f",&number);
```



```
  temp=number+0.5;
  printf("%g can be rounded to %g",number,floor(temp));
}
```

12.

```
/* To round a floating-point number to its nearest tenth and hundredth */
#include<stdio.h>
main()
{ float number;
  int tenth, hundredth;
  printf("Enter the Number ");
  scanf("%f",&number);
  tenth=((int)(number+5)/10)*10;
  hundredth=((int)(number+50)/100)*100;
  printf("%g can be rounded to the nearest tenth as %d\n",number,tenth);
  printf("%g can be rounded to the nearest hundredth as %d" , number,hundredth);
}
```

13.

```
/* To round an integer to its nearest tenth and hundredth */
#include<stdio.h>
main()
{ int number,tenth,hundredth;
  printf("Enter the Number ");
  scanf("%d",&number);
  tenth=((number+5)/10)*10;
  hundredth=((number+50)/100)*100;
  printf("%d can be rounded to the nearest
       tenth as %d\n",number,tenth);
  printf("%d can be rounded to the nearest
       hundredth as %d",number,hundredth);
}
```

15.

```
/* To find the denomination of an amount*/
#include<stdio.h>
main()
{ int amt,temp,n2000,n1000,n500,n200,n100;
  int n50,n20,n10,n5,n2,n1,total;
  printf("Enter the Amount ");
  scanf("%d",&amt);
  temp = amt;
  n2000 = amt/1000;
  amt = amt%2000;
  n1000 = amt/1000;
```

```
    amt = amt%1000;
    n500 = amt/500;
    amt = amt%500;
    n200 = amt/200;
    amt = amt%200;
    n100 = amt/100;
    amt = amt%100;
    n50 = amt/50;
    amt = amt%50;
    n20 = amt/20;
    amt = amt%20;
    n10 = amt/10;
    amt = amt%10;
    n5 = amt/5;
    amt = amt%5;
    n2 = amt/2;
    n1 = amt%2;
    total = n2000+n1000+n500+n200+n100+n50+n20+n10+n5+n2+n1;
    printf("Rs. %d contains %d 2000 Rupee notes\n",temp,n2000);
    printf("Rs. %d contains %d 1000 Rupee notes\n",temp,n1000);
    printf("Rs. %d contains %d  500 Rupee notes\n",temp,n500);
    printf("Rs. %d contains %d  200 Rupee notes\n",temp,n200);
    printf("Rs. %d contains %d  100 Rupee notes\n",temp,n100);
    printf("Rs. %d contains %d   50 Rupee notes\n",temp,n50);
    printf("Rs. %d contains %d   20 Rupee notes\n",temp,n20);
    printf("Rs. %d contains %d   10 Rupee notes\n",temp,n10);
    printf("Rs. %d contains %d    5 Rupee notes\n",temp,n5);
    printf("Rs. %d contains %d    2 Rupee notes\n",temp,n2);
    printf("Rs. %d contains %d    1 Rupee  notes\n",temp,n1);
    printf("Rs. %d contains a total of %d notes",temp,total);
}
```

16.

```
/* To display the sum of the individual digits of a four digit number */
#include<stdio.h>
main()
{  int number,temp,d1,d2,d3,d4,sum;
   printf("Enter a Number ");
   scanf("%d",&number);
   temp = number;
   d4 = number/1000;
   number = number%1000;
   d3 = number/100;
   number = number%100;
   d2 = number/10;
   d1 = number%10;
```



```
   sum = d4+d3+d2+d1;
   printf("Sum of the individual digits of %d is %d", temp,sum);
}
```

17.

```
/* To display the mean and standard deviation of three numbers */
#include<stdio.h>
#include<math.h>
main()
{ float a,b,c,m,sd;
  printf("Enter a Three Numbers\n");
  printf("First Number ? ");
  scanf("%f",&a);
  printf("Second Number ? ");
  scanf("%f",&b);
  printf("Third Number ? ");
  scanf("%f",&c);
  m=(a+b+c)/3.0;
  sd = sqrt((pow((a-m),2)+pow((b-m),2)+pow((c-m),2))/3.0);
  printf("Average of %g, %g and %g is %g\n",a,b,c,m);
  printf("Standard deviation of %g, %g and %g is %g",a,b,c,sd);
}
```

21.

```
/* To display whether an integer is even or odd */
#include<stdio.h>
main()
{    int number;
     printf("Enter an integer number : ");
     scanf("%d",&number);
     if(number%2 == 0)
          printf("%d is an Even Number",number);
     else
          printf("%d is an Odd Number",number);
}
```

23.

```
/* To display the largest of three numbers */
#include<stdio.h>
main()
{    float num1,num2,num3,big;
     printf("Enter Three Numbers\n");
     printf("First Number  : ",&num1);
     scanf("%f",&num1);
     printf("Second Number : ",&num2);
```

```
    scanf("%f",&num2);
    printf("Third Number : ",&num3);
    scanf("%f",&num3);
    big = num1;
    if(big < num2)
        big=num2;
    if(big < num3)
        big=num3;
    printf("Largest among %g, %g and %g is %g", num1,num2,num3,big);
}
```

25.

```
/* To display the whole numbers from 1 to 100 */
#include<stdio.h>
main()
{   int num;
  printf("Whole number from 1 to 100 are : \n");
  for(num=1;num<=100;++num)
    printf("%d\t",num);
}
```

32.

```
/* to display the even numbers up to a given number */
#include<stdio.h>
main()
{ int number,num;
  printf("even numbers up to which number ? ");
  scanf("%d",&number);
  printf("the even numbers from 2 to %d are : \n", number);
  for(num=2;num<=number;num+=2)
        printf("%d\t",num);
}
```

35.

```
/* to display the first 'n' odd numbers */
#include<stdio.h>
main()
{ int n,num=1,count;
  printf("how many odd numbers ? ");
  scanf("%d",&n);
  printf("the first %d odd numbers are :\n",n);
for(count=0;count<n;++count)
  { printf("%d\t",num);
    num+=2;
  }
```



```
}
```

38.

```
/* to display the first 'n' even numbers in reverse order */
#include<stdio.h>
main()
{ int num;
  printf("how many even numbers ? ");
  scanf("%d",&num);
  printf("first %d even numbers in reverse order is : \n",num);
  num=num*2;
  for(;num>1;num-=2)
     printf("%d\t",num);
}
```

42.

```
/* To display the multiples of 5 within a given range.
   Both the lower and upper bounds are excluded */
#include<stdio.h>
main()
{    int lower,upper,num,temp;
     printf("Enter the Lower bound : ");
     scanf("%d",&lower);
     printf("Enter the Upper bound : ");
     scanf("%d",&upper);
     if(lower>upper)
     {    temp = lower;
          lower = upper;
          upper = temp;
     }
     printf("The multiples of 5 between %d and %d are :\n", lower,upper);
     if (lower%5 == 0)
          lower = lower+5;
     else
          lower = lower+(5-(lower%5));
     for(;lower<upper;lower+=5)
               printf("%d\t",lower);
}
```

45.

```
/* To display the first 'n' odd numbers. Also display their sum and average */
#include<stdio.h>
main()
{  int n,num=1,count=0;
   float sum=0.0,average;
```

```
    printf("How many Odd numbers ? ");
    scanf("%d",&n);
    printf("The first %d Odd numbers are :\n",n);
    for(;count<n;++count)
    { printf("%d\t",num);
      sum+=num;
      num+=2;
    }
    average = sum/n;
    printf("\nSum of the first %d Odd numbers is %g",n,sum);
    printf("\nAverage of the first %d Odd numbers is %g", n,average);
}
```

49.

```
/* To display the Even and Odd numbers within a range. Display the numbers separately.
Also display their sum and average. Both the lower and upper bounds are exclusive */
#include<stdio.h>
main()
{   int lower,upper,even,odd,count,temp;
    float evensum=.0,oddsum=0.0,evenavg,oddavg;
    printf("Enter the Lower bound : ");
    scanf("%d",&lower);
    printf("Enter the Upper bound : ");
    scanf("%d",&upper);
    if(lower>upper)
    {   temp = lower;
        lower = upper;
        upper = temp;
    }
    if (lower%2 == 0)
        even = lower+2;
    else
        even = lower+1;
    printf("Even numbers between %d and %d are :\n",lower,upper);
    for(count=0;even<upper; even+=2, ++count)
    {   printf("%d\t",even);
        evensum+=even;
    }
    evenavg=evensum/count;
    printf("\nSum of the even numbers from %d to %d is %g",  lower,upper, evensum);
    printf("\nAverage of the even numbers from %d to %d is %g\n\n",
        lower,upper,evenavg);
    if (lower%2 == 1)
        odd = lower+2;
    else
        odd = lower+1;
```



```
    printf("Odd numbers between %d and %d are :\n", lower,upper);
    for(count=0;odd<upper;odd+=2,++count)
    {   printf("%d\t",odd);
        oddsum+=odd;
    }
    oddavg=oddsum/count;
    printf("\nSum of the odd numbers from %d to %d is %g", lower,upper,oddsum);
    printf("\nAverage of the odd numbers from %d to %d is %g", lower,upper,oddavg);
}
```

52.

```
/* To display the first 'n' prime numbers */
#include<stdio.h>
#include<math.h>
main()
{ int num,i,n,flag,count=0;
  printf("How many prime numbers ? ");
  scanf("%d",&n);
  printf("First %d Prime Numbers are:\n",n);
  for(num=2;count<=n;++num)
  { flag = 1;
    for(i= 2;i<=sqrt(num);++i)
      if( num%i == 0)
      { flag=0;
        break;
      }
    if(flag)
    { printf("%d\t",num);
      ++count;
    }
  }
}
```

55.

```
/* To display the Armstrong numbers up to a given number using for loops */
#include<stdio.h>
#include<math.h>
main()
{ int num,n,sum,digit,temp;
  printf("Up to which number ? ");
  scanf("%d",&n);
  printf("Armstrong Numbers Up to %d are:\n",n);
  for(num=1;num<=n;++num)
  { for(temp=num,sum=0;temp>0; temp/=10)
    { digit=temp % 10;
```

```
        sum+=pow(digit,3);
      }
      if(num==sum)
        printf("%d\t",num);
    }
}
```

or

```
/* To display the Armstrong numbers up to a given  number using a for loop and
a while loop */
#include<stdio.h>
#include<math.h>
main()
{ int num,n,sum,digit,temp;
  printf("Up to which number ? ");
  scanf("%d",&n);
  printf("Armstrong Numbers Up to %d are:\n",n);
  for(num=1;num<=n;++num)
  { temp=num;
    sum=0;
    while(temp>0)
    { digit=temp % 10;
      sum+=pow(digit,3);
      temp/=10;
    }
    if(num==sum) printf("%d\t",num);
  }
}
```

94.

```
/* To find the best matches for the expression 3x+2y-z=0 within the range x,y,z=1 to 10 */
#include<stdio.h>
main()
{ int x,y,z;
  printf("The best matches are :\n");
  for(x=1;x<=10;++x)
    for(y=1;y<=10;++y)
      for(z=1;z<=10;++z)
        if(3*x+2*y-z==0)
          printf("x = %d\ty = %d\tz = %d\n",x,y,z);
}
```

99.

```
/* To generate the pattern */
#include<stdio.h>
#include<conio.h>
main()
{ int i=1,j=1,n,x=3,y=40;
  printf("How many lines ? ");
  scanf("%d",&n);
  gotoxy(y,x);
  for(;i<=n;++i,++x,y-=2)
  { gotoxy(y,x);
    for(j=1;j<=i;++j)
      printf("%4d",j);
    printf("\n");
  }
}
```

111.

```
/* To draw the reliability graph */
#include<stdio.h>
```

```
#include<math.h>
#define LAMBDA 0.001
main()
{ float t=0,r;
  int i,R;
  for(i=1;i<50;i++) printf("-");
  for(t=0;t<=2000;t+=200)
  { r=exp(-LAMBDA*t);
    R=r*50;/* applying a scaling factor for drawing */
    printf("\n|");
    for(i=1;i<=R;i++) printf("*");
    printf(" (r=%f)",r);/* Displaying value as legend */
  }
}
```

114.

```
/* To display the power table */
#include<stdio.h>
#include<math.h>
main()
{ int x,n;
  printf("x\tn=1\t2\t3\t4\t5");
  for(x=1;x<=5;x++)
  { printf("\n%2d  ",x);
    for(n=1;n<=5;n++)
      printf("%6.0f ",pow(x,n));
    printf("\n");
  }
}
```