Unit 4 Control Statements and Decision Making

Structure:

4.1 Introduction
Objectives
4.2 The **goto** statement
4.3 The **if** statement
The **if-else** statement
Nesting of **if** statements
4.4 The conditional expression
4.5 The **switch** statement
4.6 The **while** loop
4.7 The **do…while** loop
4.8 The **for** loop
The nesting of **for** loops
4.9 The **break** statement and **continue** statement
4.10 Summary
4.11 Terminal Questions
4.12 Answers to Self Assessment Questions
4.13 Answers to Terminal Questions
4.14 Exercises

4.1 Introduction

In the previous unit, you studied about the data types that are supported in C and the types of Input/output operators available in C. This unit will enable you to use the various types of control statements for making a decision in C. Statements are the “steps" of a program. Most statements compute and assign values or call functions, but we will eventually meet several other kinds of statements as well. By default, statements are executed in sequence, one after another. We can, however, modify that sequence by using *control flow constructs* such that a statement or group of statements is executed only if some condition is true or false. This involves a kind of decision making to see whether a particular condition has occurred or not and then direct the computer to execute certain statements accordingly.

C language possesses such decision making capabilities and supports the following statements known as the *control* or *decision making* statements.

- **if** statement
- **switch** statement
- **goto** statement
- conditional operator statement

You will also study about the loops in this unit. Loops generally consist of two parts: one or more control expressions which control the execution of the loop, and the body, which is the statement or set of statements which is executed over and over.

As far as C is concerned, a true/false condition can be represented as an integer. (An integer can represent many values; here we care about only two values: “true" and “false." The study of mathematics involving only two values is called Boolean algebra, after George Boole, a mathematician who refined this study.) In C, “false" is represented by a value of 0 (zero), and “true" is represented by any value that is nonzero. Since there are many nonzero values (at least 65,534, for values of type int), when we have to pick a specific value for “true," we'll pick 1.

Do…while loop is used in a situation where we need to execute the body of the loop before the test is performed. The for loop is used to execute the body of the loop for a specified number of times. The break statement is used to exit any loop.

Objectives:

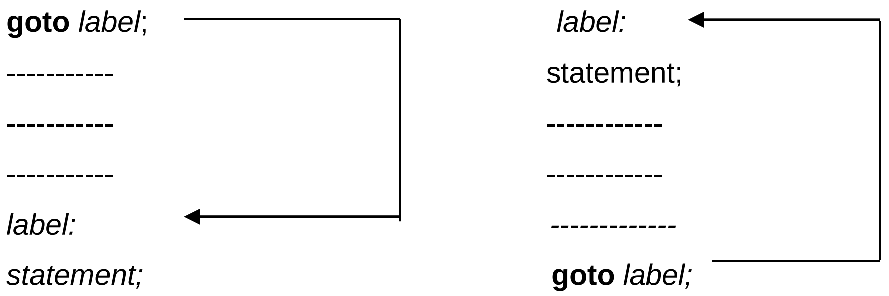
After studying this unit, you should be able to:

- control the flow of execution of statements using two-way decision and multipath decision.
- branch unconditionally from one point to another in the program.
- evaluate the conditional expressions.
- repeat the execution of statements by checking the condition before the loop body is executed and by checking the condition at the end of the loop.
- exit from the loop depending on some condition.
- break the current iteration and continue with next iteration of loop.

4.2 The *goto* statement

C supports the **goto** statement to branch unconditionally from one point to another in the program. Although it may not be essential to use the **goto** statement in a highly structured language like C, there may be occasions when the use of **goto** might be desirable.

The **goto** requires a *label* in order to identify the place where the branch is to be made. A *label* is any valid variable name, and must be followed by a colon. The *label* is placed immediately before the statement where the control is to be transferred. The general forms of **goto** and *label* statements are shown below:



```
goto label;
-----------
-----------
-----------
label:
statement;
```

(i) Forward jump

```
label:
statement;
------------
------------
-------------
goto label;
```

(ii) Backward jump

The *label* can be anywhere in the program either before the **goto** or after the **goto** *label;* statement.

During execution of the program when a statement like

goto first;

is met, the flow of control will jump to the statement immediately following the label **first.** This happens unconditionally.

Note that a **goto** breaks the normal sequential execution of the program. If the *label* is before the statement **goto** *label;* a loop will be formed and some statements will be executed repeatedly. Such a jump is known as a *backward jump.* On the other hand, if the *label* is placed after the **goto** *label;* some statements will be skipped and the jump is known as the *forward jump.*

A **goto** is often used at the end of a program to direct the control to go to the input statement, to read further data. Consider the following example:

Program 4.1: Program showing unconditional branching

```
main()
{
        double a, b;
        read:
        printf("enter the value of a\n");
        scanf("%f", &a);
        if (a<0) goto read;
        b=sqrt(a);
        printf("%f %f \n",a, b);
        goto read;
}
```

This program is written to evaluate the square root of a series of numbers read from the terminal. The program uses two **goto** statements, one at the end, after printing the results to transfer the control back to the input statements and the other to skip any further computation when the number is negative.

Due to the unconditional **goto** statement at the end, the control is always transferred back to the input statement. In fact, this program puts the computer in a permanent loop known as an *infinite* loop.

Self Assessment Questions

1. The goto requires a _________ in order to identify the place where the branch is to be made.
2. *goto* is an unconditional branching statement. (True/False)

4.3 The *if* statement

The simplest way to modify the control flow of a program is with an if statement, which in its simplest form looks like this:

```
if(x > max)
        max = x;
```

Even if you didn't know any C, it would probably be pretty obvious that what happens here is that if x is greater than max, x gets assigned to max. (We'd use code like this to keep track of the maximum value of x we'd seen--for each new x, we'd compare it to the old maximum value max, and if the new value was greater, we'd update max.)

More generally, we can say that the syntax of an if statement is:

```
        if( expression )
                statement
```

where *expression* is any expression and *statement* is any statement.

What if you have a series of statements, all of which should be executed together or not at all depending on whether some condition is true? The answer is that you enclose them in braces:

```
        if( expression )
                {
                statement 1;
                statement 2;
                statement n;
                }
```

As a general rule, anywhere the syntax of C calls for a statement, you may write a series of statements enclosed by braces. (You do not need to, and should not, put a semicolon after the closing brace, because the series of statements enclosed by braces is not itself a simple expression statement.)

Program 4.2: Program to calculate the absolute value of an integer

```
# include < stdio.h >
void main ( )
{
int number;
printf (“Type a number:”);
scanf (“%d”, & number);
if (number < 0) /* check whether the number is a negative number */
        number = - number; /* If it is negative then convert it into positive. */
printf (“The absolute value is % d \n”, number);
}
```

4.3.1 The *if-else* statement

An **if** statement may also optionally contain a second statement, the **“else** clause," which is to be executed if the condition is not met. Here is an example:

```
        if(n > 0)
        average = sum / n;
else    {
        printf("can't compute average\n");
        average = 0;
        }
```

The first statement or block of statements is executed if the condition *is* true, and the second statement or block of statements (following the keyword else) is executed if the condition is *not* true. In this example, we can compute a meaningful average only if n is greater than 0; otherwise, we print a message saying that we cannot compute the average. The general syntax of an **if** statement is therefore

```
if( expression )
        statement(s)
else
        statement(s)
```

(if there are more than one statements, they should be enclosed within braces).

Program 4.3: To find whether a number is negative or positive

```
#include < stdio.h >
void main ( )
{
int num;
printf (“Enter the number”);
scanf (“%d”, &num);
if (num < 0)
printf (“The number is negative”)
else
        printf (“The number is positive”);
 }
```

4.3.2 Nesting of *if* statements

It's also possible to nest one **if** statement inside another. (For that matter, it's in general possible to nest any kind of statement or control flow construct within another.) For example, here is a little piece of code which decides roughly which quadrant of the compass you're walking into, based on an x value which is positive if you're walking east, and a y value which is positive if you're walking north:

```
if(x > 0)
        {
        if(y > 0)
                printf("Northeast.\n");
        else    printf("Southeast.\n");
        }
else    {
        if(y > 0)
                printf("Northwest.\n");
        else    printf("Southwest.\n");
        }
```

When you have one **if** statement (or loop) nested inside another, it's a very good idea to use explicit braces {}, as shown, to make it clear (both to you and to the compiler) how they're nested and which **else** goes with which **if**. It's also a good idea to indent the various levels, also as shown, to make the code more readable to humans. Why do both? You use indentation to make the code visually more readable to yourself and other humans, but the compiler doesn't pay attention to the indentation (since all whitespace is essentially equivalent and is essentially ignored). Therefore, you also have to make sure that the punctuation is right.

Here is an example of another common arrangement of **if** and **else.** Suppose we have a variable **grade** containing a student's numeric grade, and we want to print out the corresponding letter grade. Here is the code that would do the job:

```
if(grade >= 90)
        printf("A");
else if(grade >= 80)
        printf("B");
else if(grade >= 70)
        printf("C");
else if(grade >= 60)
        printf("D");
else    printf("F");
```

What happens here is that exactly one of the five **printf** calls is executed, depending on which of the conditions is true. Each condition is tested in turn, and if one is true, the corresponding statement is executed, and the rest are skipped. If none of the conditions is true, we fall through to the last one, printing “F".

In the cascaded **if/else/if/else/...** chain, each **else** clause is another **if** statement. This may be more obvious at first if we reformat the example, including every set of braces and indenting each **if** statement relative to the previous one:

```
if(grade >= 90)
        {
        printf("A");
        }
else    {
        if(grade >= 80)
                {
                printf("B");
                }
        else    {
                if(grade >= 70)
                        {
                        printf("C");
                        }
                else    {
                        if(grade >= 60)
                                {
                                printf("D");
                                }
                        else    {
                                printf("F");
                                }
                        }
                }
        }
```

By examining the code this way, it should be obvious that exactly one of the **printf** calls is executed, and that whenever one of the conditions is found true, the remaining conditions do not need to be checked and none of the later statements within the chain will be executed. But once you've convinced yourself of this and learned to recognize the idiom, it's generally preferable to arrange the statements as in the first example, without trying to indent each successive **if** statement one tabstop further out.

Program 4.4: Program to print the largest of three numbers

```
#include<stdio.h>
main()
{
int a,b,c,big;
printf (“Enter three numbers”);
scanf (“%d %d %d”, &a, &b, &c);
if (a>b) // check whether a is greater than b if true then
      if(a>c) // check whether a is greater than c
            big = a ; // assign a to big
      else big = c ; // assign c to big
else if (b>c) // if the condition (a>b) fails check whether b is greater than c
          big = b ; // assign b to big
else  big = c ; // assign C to big
printf (“Largest of %d,%d&%d = %d”, a,b,c,big);
}
```

Self Assessment Questions:

3. The series of statements enclosed by braces after the expression in simple *if* statement is itself a simple expression statement. (True/False)
4. In the cascaded if/else/if/else/... chain, each else clause is another _______ statement.

4.4 The conditional expression

The conditional operator (?:) takes three operands. It tests the result of the first operand and then evaluates one of the other two operands based on the result of the first. Consider the following example:

E1 ? E2 : E3

If expression *E1* is nonzero (true), then *E2* is evaluated, and that is the value of the conditional expression. If *E1* is 0 (false), *E3* is evaluated, and that is the value of the conditional expression. Conditional expressions associate from right to left. In the following example, the conditional operator is used to get the minimum of x and y:

```
a = (x < y) ? x : y;   /* a= min(x, y)  */
```

There is a sequence point after the first expression (*E1*). The following example's result is predictable, and is not subject to unplanned side effects:

```
i++ > j ? y[i] : x[i];
```

The conditional operator does not produce a lvalue. Therefore, a statement such as

a ? x : y = 10 is not valid.

4.5 The *switch* statement

The **switch case** statements are a substitute for long **if** statements that compare a variable to several "integral" values ("integral" values are simply values that can be expressed as an integer, such as the value of a **char**). The basic format for using **switch case** is outlined below. The value of the variable given into **switch** is compared to the value following each of the cases, and when one value matches the value of the variable, the computer continues executing the program from that point.

```
switch ( <variable> ) {
case this-value:
  Code to execute if <variable> == this-value
  break;
case that-value:
  Code to execute if <variable> == that-value
  break;
...
default:
  Code to execute if <variable> does not equal the value following any of the
cases
  break;
}
```

The condition of a **switch** statement is a value. The **case** says that if it has the value of whatever is after that **case** then do whatever follows the colon. The break is used to **break** out of the **case** statements. **break** is a keyword that breaks out of the code block, usually surrounded by braces, which it is in. In this case, **break** prevents the program from falling through and executing the code in all the other **case** statements. An important thing to note about the **switch** statement is that the **case** values may only be constant integral expressions. It isn't legal to use **case** like this:

```
int a = 10;
int b = 10;
int c = 20;
switch ( a ) {
case b:
  /* Code */
  break;
case c:
  /* Code */
  break;
default:
  /* Code */
  break;
}
```

The **default** case is optional, but it is wise to include it as it handles any unexpected cases. It can be useful to put some kind of output to alert you to the code entering the **default** case if you don't expect it to. Switch statements serve as a simple way to write long **if** statements when the requirements are met. Often it can be used to process input from a user.

Example: Below is a sample program, in which not all of the proper functions are actually declared, but which shows how one would use switch in a program.

```
#include <stdio.h>
void playgame();
void loadgame();
void playmultiplayer();
int main()
```

```
{
    int input;
    printf( "1. Play game\n" );
    printf( "2. Load game\n" );
    printf( "3. Play multiplayer\n" );
    printf( "4. Exit\n" );
    printf( "Selection: " );
    scanf( "%d", &input );
    switch ( input ) {
        case 1:            /* Note the colon, not a semicolon */
            playgame();
            break;
        case 2:
            loadgame();
            break;
        case 3:
            playmultiplayer();
            break;
        case 4:
            printf( "Thanks for playing!\n" );
            break;
        default:
            printf( "Bad input, quitting!\n" );
            break;
    }
    getchar();
}
```

This program will compile, but cannot be run until the undefined functions are given bodies, but it serves as a model (albeit simple) for processing input. If you do not understand this then try mentally putting in **if** statements for the **case** statements. **default** simply skips out of the **switch case** construction and allows the program to terminate naturally. If you do not like that, then you can make a loop around the whole thing to have it wait for valid input. You could easily make a few small functions if you wish to test the code.

Self Assessment Questions:

5. The conditional operator does not produce a lvalue. (True/False)
6. The condition of a **switch** statement is a _______.
7. Switch statement is an unconditional branching statement. (True/False)

4.6 The *while* loop

Loops generally consist of two parts: one or more *control expressions* which control the execution of the loop, and the *body*, which is the statement or set of statements which is executed over and over.

The most basic *loop* in C is the **while** loop. A **while** loop has one control expression, and executes as long as that expression is true. Here before executing the body of the loop, the condition is tested. Therefore it is called an **entry-controlled** loop. The following example repeatedly doubles the number 2 (2, 4, 8, 16, ...) and prints the resulting numbers as long as they are less than 1000:

```
int x = 2;

while(x < 1000)
        {
        printf("%d\n", x);
        x = x * 2;
        }
```

(Once again, we've used braces {} to enclose the group of statements which are to be executed together as the body of the loop.)

The general syntax of a **while** loop is

```
while( expression )
        statement(s)
```

A **while** loop starts out like an **if** statement: if the condition expressed by the *expression* is true, the *statement* is executed. However, after executing the statement, the condition is tested again, and if it's still true, the statement is executed again. (Presumably, the condition depends on some value which is changed in the body of the loop.) As long as the condition remains true, the body of the loop is executed over and over again. (If the condition is false right at the start, the body of the loop is not executed at all.)

As another example, if you wanted to print a number of blank lines, with the variable **n** holding the number of blank lines to be printed, you might use code like this:

```
        while(n > 0)
                {
                printf("\n");
                n = n - 1;
                }
```

After the loop finishes (when control “falls out" of it, due to the condition being false), n will have the value 0.

You use a **while** loop when you have a statement or group of statements which may have to be executed a number of times to complete their task. The controlling expression represents the condition “the loop is not done" or “there's more work to do." As long as the expression is true, the body of the loop is executed; presumably, it makes at least some progress at its task. When the expression becomes false, the task is done, and the rest of the program (beyond the loop) can proceed. When we think about a loop in this way, we can see an additional important property: if the expression evaluates to “false" before the very first trip through the loop, we make *zero* trips through the loop. In other words, if the task is already done (if there's no work to do) the body of the loop is not executed at all. (It's always a good idea to think about the “boundary conditions" in a piece of code, and to make sure that the code will work correctly when there is no work to do, or when there is a trivial task to do, such as sorting an array of one number. Experience has shown that bugs at boundary conditions are quite common.)

Program 4.5: Program to find largest of n numbers

```
main()
{
int num, large, n, i;
clrscr();
printf("enter number of numbers \n");
scanf(“%d”,&n);
large=0;
i=0;
while(i<n)
{
printf("\n enter number ");
scanf("%d", &num);
  if(large<num)
  large=num;
  i++;
}
printf("\n large = %d", large);
}
```

Program 4.6: Program to evaluate sine series sin(x)=x-x^3/3!+x^5/5!-x^7/7!+----- depending on accuracy

```
# include<stdio.h>
# include <math.h>
void main()
{
  int n, i=1,count;
  float acc, x, term, sum;
  printf("enter the angle\n");
  scanf("%d", &x);
  x=x*3.1416/180.0;
  printf("\nenter the accuracy)";
  scanf("%f", &acc);
sum=x;
  term=x;
  while ((fabs(term))>acc)
  {
     term=-term*x*x/((2*i)*(2*i+1));
     sum+=term;
     i++;
     }
     printf"\nsum of sine series is %f", sum);
     }
```

Self Assessment Questions

8. A ______________ loop starts out like an **if** statement .
9. *while* is an entry-controlled loop. (True/False)

4.7 The do…while loop

The **do…while** loop is used in a situation where we need to execute the body of the loop before the test is performed. Therefore, the body of the loop may not be executed at all if the condition is not satisfied at the very first attempt. Where as while loop makes a test of condition before the body of the loop is executed.

For above reasons **while** loop is called an **entry-controlled loop** and **do..while** loop is called an **exit-controlled loop**.

do while loop takes the following form:

```
do
{
      Body of the loop
}
while  ( expression);
```

On reaching the **do** statement , the program proceeds to evaluate the body of the loop first. At the end of the loop, the conditional expression in the **while** statement is evaluated. If the expression is true, the program continues to evaluate the body of the loop once again. This process continues as long as the expression is true. When the expression becomes false, the loop will be terminated and the control goes to the statement that appears immediately after the **while** statement.

On using the **do** loop, the body of the loop is always executed at least once irrespective of the expression.

Program 4.7: A program to print the multiplication table from 1 x 1 to 10 x 10 as shown below using do-while loop.

```
1     2     3     4     ……………………    10
2     4     6     8     ……………………    20
3     6     9     12    ……………………    30
4                       ……………………    40
.                                        .
.                                        .
.                                        .
10                                      100
```

```
// Program to print multiplication table
main()
{
  int rowmax=10,colmax=10,row,col,x;
  printf("    Multiplication table\n");
  printf("......................................\n");
  row=1;
do
  {
     col=1;
     do
     {
       x=row*col;
       printf("%4d", x);
       col=col+1;
      }
      while (col<=colmax);
      printf("\n");;
      row=row+1;
  }
  while(row<=rowmax);
Printf("...................................................................................................\
n");
}
```

Self Assessment Questions:

10. On using the ________, the body of the loop is always executed at least once irrespective of the expression.
11. *do…while* is an entry-controlled loop. (True/False)

4.8 The *for* loop

The **for** loop is used to repeat the execution of set of statements for a fixed number of times. The **for** loop is also an **entry-controlled loop.**

Generally, the syntax of a for loop is

```
for(expr1; expr2; expr3) statement(s)
```

(Here we see that the **for** loop has three control expressions. As always, the statement can be a brace-enclosed block.)

Many loops are set up to cause some variable to step through a range of values, or, more generally, to set up an initial condition and then modify some value to perform each succeeding loop as long as some condition is true. The three expressions in a **for** loop encapsulate these conditions: expr1 sets up the initial condition, expr 2 tests whether another trip through the loop should be taken, and expr3 increments or updates things after each trip through the loop and prior to the next one. Consider the following :

```
for (i = 0; i < 10; i = i + 1)
                printf("i is %d\n", i);
```

In the above example, we had i = 0 as expr1, i < 10 as expr2 , i = i + 1 as expr3, and the call to **printf** as statement, the body of the loop. So the loop began by setting i to 0, proceeded as long as i was less than 10, printed out i's value during each trip through the loop, and added 1 to i between each trip through the loop.

When the compiler sees a for loop, first, expr1 is evaluated. Then, expr2 is evaluated, and if it is true, the body of the loop (statement) is executed. Then, expr3 is evaluated to go to the next step, and expr2 is evaluated again, to see if there is a next step. During the execution of a for loop, the sequence is:

expr1

expr2

statement

expr3

expr2

statement

expr3

...

expr2

statement

expr3

expr2

The first thing executed is expr1. expr3 is evaluated after every trip through the loop. The last thing executed is always expr2, because when expr2 evaluates false, the loop exits.

All three expressions of a for loop are optional. If you leave out expr1, there simply is no initialization step, and the variable(s) used with the loop had better have been initialized already. If you leave out expr2, there is no test, and the default for the for loop is that another trip through the loop should be taken (such that unless you break out of it some other way, the loop runs forever). If you leave out expr3, there is no increment step.

The semicolons separate the three controlling expressions of a for loop. (These semicolons, by the way, have nothing to do with statement terminators.) If you leave out one or more of the expressions, the semicolons remain. Therefore, one way of writing a deliberately infinite loop in C is

```
for(;;)
        ...
```

It's also worth noting that a for loop can be used in more general ways than the simple, iterative examples we've seen so far. The “control variable" of a for loop does not have to be an integer, and it does not have to be incremented by an additive increment. It could be “incremented" by a multiplicative factor (1, 2, 4, 8, ...) if that was what you needed, or it could be a floating-point variable, or it could be another type of variable which we haven't met yet which would step, not over numeric values, but over the elements of an array or other data structure. Strictly speaking, a **for loop** doesn't have to have a “control variable" at all; the three expressions can be anything, although the loop will make the most sense if they are related and together form the expected initialize, test, increment sequence.

The powers-of-two example using **for** is:

```
int x;
for(x = 2; x < 1000; x = x * 2)
        printf("%d\n", x);
```

There is no earth-shaking or fundamental difference between the **while** and for **loops**. In fact, given the general for loop

```
for(expr1; expr2; expr3)
        statement
```

you could usually rewrite it as a while loop, moving the initialize and increment expressions to statements before and within the loop:

```
expr1;
while(expr2)
        {
        statement
        expr3;
        }
```

Similarly, given the general while loop

```
while(expr)
        statement
```

you could rewrite it as a for loop:

```
for(; expr; )
        statement
```

Another contrast between the **for** and **while** loops is that although the test expression (expr2) is optional in a **for** loop, it is required in a **while** loop. If you leave out the controlling expression of a **while** loop, the compiler will complain about a syntax error. (To write a deliberately infinite **while** loop, you have to supply an expression which is always nonzero. The most obvious one would simply be while(1) .)

If it's possible to rewrite a **for** loop as a **while** loop and vice versa, why do they both exist? Which one should you choose? In general, when you choose a for loop, its three expressions should all manipulate the same variable or data structure, using the initialize, test, increment pattern. If they don't manipulate the same variable or don't follow that pattern, wedging them into a **for** loop buys nothing and a **while** loop would probably be clearer. (The reason that one loop or the other can be clearer is simply that, when you see a **for** loop, you expect to see an idiomatic initialize/ test/increment of a single variable, and if the **for** loop you're looking at doesn't end up matching that pattern, you've been momentarily misled.)

Program 4.8: A Program to find the factorial of a number

```
void main()
{
int M,N;
long int F=1;
clrscr();
printf(“enter the number\n”)";
scanf(“%d”,&N);
if(N<=0)
F=1;
else
{
for(M=1;M<=N;M++)
F*=M;
}
printf(“the factorial of the number  is %ld”,F);
getch();
}
```

Self Assessment Questions

12. **for** loop is an **exit-controlled** loop. (True/False)
13. The “control variable" of a **for** loop does not have to be an integer. (True/False)

4.8.1 The Nesting of for loops

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

```
        ………
        ………
        for(i=1;i<10;i++)
        {
          …….
          …….
          for(j=1;j<5;j++)
          {
            ……
            ……
        }
        …….
        …….
    }
    ………
    ………
```

4.9 The *break* statement and continue statement

The purpose of ***break*** statement is to break out of a loop (**while**, **do while**, or **for** loop) or a **switch** statement. When a **break** statement is encountered inside a loop, the loop is immediately exited and the program continues with the statement immediately following the loop. When the loops are nested, the **break** would only exit from the loop containing it. That is, the **break** would exit only a single loop.

Syntax : break;

Program 4.9: Program to illustrate the use of *break* statement.

```
void main ( )
{
  int x;
  for (x=1; x<=10; x++)
  {
      if (x==5)
      break; /*break loop only if x==5 */
      printf("%d", x);
 }
  printf ("\nBroke out of loop");
  printf( "at x =%d");
 }
```

The above program displays the numbers from 1to 4 and prints the message "Broke out of loop when 5 is encountered.

The *continue statement*

The ***continue*** statement is used to continue the next iteration of the loop by skipping a part of the body of the loop (for, **do/while** or **while** loops). The *continue* statement does not apply to a **switch**, like a ***break*** statement.

Unlike the **break** which causes the loop to be terminated, the **continue**, causes the loop to be continued with the next iteration after skipping any statements in between.

Syntax: continue;

Program 4.10: Program to illustrate the use of *continue* statement.

```
  void main ( )   {
  int x;
  for (x=1; x<=10; x++)
          {  if (x==5)
                      continue; /* skip remaining code in loop
                         only if x == 5 */
                     printf (“%d\n”, x);
          }
  printf(“\nUsed continue to skip”);
 }
```

The above program displays the numbers from 1to 10, except the number 5.

Program 4.11: Program to sum integers entered interactively

```
#include <stdio.h>
int main(void)
{
  long num;
  long sum = 0L;        /* initialize sum to zero     */
  int status;

  printf("Please enter an integer to be summed. ");
  printf("Enter q to quit.\n");
  status = scanf("%ld", &num);
  while (status == 1)
    {
        sum = sum + num;
        printf("Please enter next integer to be summed. ");
        printf("Enter q to quit.\n");
        status = scanf("%ld", &num);
  }
  printf("Those integers sum to %ld.\n", sum);
  return 0;
}
```

4.10 Summary

In C by default, statements are executed in sequence, one after another. We can, however, modify that sequence by using ***control flow constructs.*** C language possesses decision making capabilities and supports the following statements known as the *control* or *decision making* statements: ***goto, if, switch****.* The **goto** statement is used to branch unconditionally from one point to another in the program. The simplest way to modify the control flow of a program is with an **if** statement. **switch** statements serve as a simple way to write long **if** statements when the requirements are met.

The most basic *loop* in C is the **while** loop. A **while** loop has one control expression, and executes as long as that expression is true. **do..while loop** is used in a situation where we need to execute the body of the loop before the test is performed. The **for** loop is used to execute the set of statements repeatedly for a fixed number of times. It is an entry controlled loop. **break** statement is used to exit any loop. Unlike the **break** which causes the loop to be terminated, the **continue**, causes the loop to be continued with the next iteration after skipping any statements in between.

4.11 Terminal Questions

1. Consider the following program segment:

```
x=1;
y=1;
if (n>0)
        x=x+1;
        y=y-1;
        printf("%d %d",x,y);
```

   What will be the values of x and y if n assumes a value of (a) 1 and (b) 0.

2. Rewrite the following without using compound relation:

```
if (grade<=59 && grade>=50)
second = second +1;
```

3. Write a program to check whether an input number is odd or even.

4. Write the **output** that will be generated by the following C program:

```
            void main()
    {
            int i=0, x=0;
            while (i<20)
            {
                    if (i%5 == 0)
                    {
                            x+=i;
                            printf("%d\t", i);
                    }
                i++;
          }
          printf("\nx=%d"; x);
    }
```

5. Write the output that will be generated by the following C program:

```
    void main()
     {
            int i=0, x=0;
            do
            {
                            if (i%5 == 0)
                    {
                            x++;
                            printf("%d\t", x);
                    }
                        ++i;
                    } while (i<20);
               printf("\nx=%d", x);
    }
```

4.12 Answers to Self Assessment Questions

1. label
2. true
3. false

4. if
5. true
6. value
7. false
8. while
9. true
10. do…while
11. false
12. false
13. true

4.13 Answers to Terminal Questions

1. (a) x=2, y=0

   (b) x=1, y=0

2.
```
if (grade<=59)
        if (grade>=50)
                second = second+1;
```

3.
```
void main()
    {
        int no;
        printf("enter a number\n");
        scanf("%d",&no);
        if (no%2==0)
                printf("even number\n");
        else printf("odd number\n");
  }
```

4.
```
0   5       10      15
x = 30
```

5.
```
1   2       3       4
x = 4
```

4.14 Exercises

1. Explain different types of **if** statements with examples.
2. Explain the syntax of **switch** statement with an example.
3. Write a program to check whether a given number is odd or even using **switch** statement.
4. Write a program to find the smallest of 3 numbers using **if-else** statement.
5. Write a program to find the roots of a quadratic equation.
6. Compare the following statements
   a) *while* and *do…while*
   b) *break* and *continue*
7. Write a program to compute the sum of digits of a given number using **while** loop.
8. Write a program that will read a positive integer and determine and print its binary equivalent using **do…while** loop.
9. The numbers in the sequence

   1 1 2 3 5 8 13 ………..

   are called Fibonacci numbers. Write a program using **do…while** loop to calculate and print the first n Fibonacci numbers.
10. Find errors, if any, in each of the following segments. Assume that all the variables have been declared and assigned values.

(a)

```
while (count !=10);
{
   count = 1;
   sum = sum + x;
   count = count + 1;
}
```

(b)

```
do;
total = total + value;
scanf("%f", &value);
while (value ! =999);
```

11. Write programs to print the following outputs using **for** loops.

```
(a) 1                    b)          1
    2 2                            2    2
    3 3 3                        3    3    3
    4 4 4 4                    4    4    4    4
```

12. Write a program to read the age of 100 persons and count the number of persons in the age group 50 to 60. Use **for** and **continue** statements.

13. Write a program to print the multiplication table using nested **for** loops.