UNIT 4: CONTROL STATEMENTS

UNIT STRUCTURE

4.0 OBJECTIVE

After going through this unit, you will be able to:

- Work with different control statements
- Know the appropriate use of the various control statements in programming
- Transfer the control from within the loops
- Use the goto, break and continue statements in the programs; and
- Write programs using branching, looping statements

4.1 INTRODUCTION

A program consists of a number of statements to be executed by the computer. Not many of the programs execute all their statements in sequential order from beginning to end as they appear within the program. A C program may require that a logical test be carried out at some particular point within the program. One of the several possible actions will be carried out, depending on the outcome of the logical test. This is called Branching. In the Selection process, a set of statements will be selected for execution, among the several sets available: Suppose, if there is a need of a group of statements to be executed repeatedly until some logical condition is satisfied, then looping is required in the program. These can be carried out using various control statements.

These Control statements determine the "flow of control" in a program and enable us to specify the order in which the various instructions in a program are to be executed by the computer.

4.2 DECISION – MAKING IN C

The C programming language provides several decision-making constructs:

- The if statement
- The switch statement
- The conditional operator

4.2.1 The if Statement

The C programming language provides a general decision-making capability in the form

of a language construct known as the if statement. The general format of this statement is as follows:

```
if ( expression )
     program statement
```

The if statement is used to stipulate execution of a program statement (or statements if enclosed in braces) based upon specified conditions.

```
if ( count > COUNT_LIMIT )
        printf ("Count limit exceeded\n");
```

the printf statement is executed *only* if the value of count is greater than the value of COUNT_LIMIT; otherwise, it is ignored.

Example 4.2.1: Calculating the Absolute Value of an Integer

```
// Program to calculate the absolute value of an integer
int main (void)
{
  int number;
  printf ("Type in your number: ");
  scanf ("%i", &number);
  if ( number < 0 )
        number = -number;
  printf ("The absolute value is %i\n", number);
  return 0;
}
```

Output

Type in your number: **-100**
The absolute value is 100

4.2.2 The if-else Construct

The general format of *if-else* construct is as follows:

```
if ( expression )
     program statement 1
else
    program statement 2
```

The if-else is actually just an extension of the general format of the if statement. If the result of the evaluation of *expression* is TRUE, *program statement 1*, which immediately follows, is executed; otherwise, *program statement 2* is executed. In either case, either *program statement 1* or *program statement 2* is executed, but not both.

Example 4.2.2

```
// Program to determine if a number is even or odd
#include <stdio.h>
int main ()
{
    int number_to_test, remainder;
    printf ("Enter your number to be tested: ");
    scanf ("%i", &number_to_test);
   remainder = number_to_test % 2;
   if ( remainder == 0 )
        printf ("The number is even.\n");
   else
        printf ("The number is odd.\n");
  return 0;
}
```

Output

Enter your number to be tested: **1234**
The number is even.

4.2.3 Compound Relational Tests

A *compound relational test* is simply one or more simple relational tests joined by either the *logical AND* or the *logical OR* operator.These operators are represented by the character pairs && and || (two vertical bar characters), respectively.As an example, the C statement

```
if ( grade >= 70 && grade <= 79 )
        ++grades_70_to_79;
```

increments the value of grades_70_to_79 only if the value of grade is greater than or equal to 70 *and* less than or equal to 79.
In a like manner, the statement

```
if ( index < 0 || index > 99 )
        printf ("Error - index out of range\n");
```

causes execution of the printf statement if index is less than 0 *or* greater than 99.

Example 4.2.3 : Determining if a Year Is a Leap Year

```
// Program to determines if a year is a leap year
#include <stdio.h>
int main (void)
{
  int year, rem_4, rem_100, rem_400;
  printf ("Enter the year to be tested: ");
  scanf ("%i", &year);
  rem_4 = year % 4;
  rem_100 = year % 100;
  rem_400 = year % 400;
  if ( (rem_4 == 0 && rem_100 != 0) || rem_400 == 0 )
      printf ("It's a leap year.\n");
  else
      printf ("Nope, it's not a leap year.\n");
  return 0;
}
```

Output

Enter the year to be tested: **1955**
Nope, it's not a leap year.

4.2.4 Nested if Statements

In the general format of the if statement, remember that if the result of evaluating the expression inside the parentheses is TRUE, the statement that immediately follows is executed. It is perfectly valid that this program statement be another if statement, as in the following statement:

```
if ( gameIsOver == 0 )
        if ( playerToMove == YOU )
            printf ("Your Move\n");
```

If the value of gameIsOver is 0, the following statement is executed, which is another if statement.This if statement compares the value of playerToMove against YOU. If the two values are equal, the message “Your Move” is displayed at the terminal.Therefore, the printf statement is executed only if gameIsOver equals 0 *and* playerToMove equals YOU.
In fact, this statement could have been equivalently formulated using compound relationals as follows:

```
if ( gameIsOver == 0 && playerToMove == YOU )
        printf ("Your Move\n");
```

A more practical example of “nested” if statements is if you added an else clause to the previous example, as follows:

```
if ( gameIsOver == 0 )
        if ( playerToMove == YOU )
            printf ("Your Move\n");
        else
            printf ("My Move\n");
```

Notice how the else clause is associated with the if statement that tests the value of playerToMove, and not with the if statement that tests the value of gameIsOver.The general rule is that an else clause is always associated with the last if statement that does not contain an else.
You can use braces to force a different association in those cases in which an innermost if does not contain an else, but an outer if does.The braces have the effect of “closing off” the if statement.

Thus,

```
if ( gameIsOver == 0 )
{
        if ( playerToMove == YOU )
            printf ("Your Move\n");
  }
else
        printf ("The game is over\n");
```

achieves the desired effect, with the message “The game is over” being displayed if the value of gameIsOver is not 0.

4.2.5 The else if Construct

The statement that followed an else can be any valid C program statement, it seems logical that it can be another if. Thus, in the general case, you could write

```
if ( expression 1 )
   program statement 1
else
    if ( expression 2 )
        program statement 2
    else
        program statement 3
```

which effectively extends the if statement from a two-valued logic decision to a threevalued logic decision.You can continue to add if statements to the else clauses, in the manner just shown, to effectively extend the decision to an *n*-valued logic decision.

Example 4.2.5 : Evaluating Simple Expressions

```
/* Program to evaluate simple expressions of the form
number operator number */
#include <stdio.h>
int main (void)
{
   float value1, value2;
   char operator;
   printf ("Type in your expression.\n");
   scanf ("%f %c %f", &value1, &operator, &value2);
   if ( operator == '+' )
        printf ("%.2f\n", value1 + value2);
   else if ( operator == '-' )
        printf ("%.2f\n", value1 - value2);
   else if ( operator == '*' )
        printf ("%.2f\n", value1 * value2);
   else if ( operator == '/' )
        if ( value2 == 0 )
            printf ("Division by zero.\n");
        else
            printf ("%.2f\n", value1 / value2);
   else
        printf ("Unknown operator.\n");
   return 0;
}
```

Output

Type in your expression.
123.5 + 59.3
182.80

Output (Rerun)

Type in your expression.
198.7 / 0
Division by zero.

Output (Second Rerun)

Type in your expression.

125 $ 28
Unknown operator.

4.2.6 The switch Statement

The general format of a switch statement is

```
switch ( expression )
{
   case value1:
          program statement
          program statement
          ...
          break;
   case value2:
          program statement
          program statement
          ...
          break;
   ...
   case valuen:
         program statement
         program statement
         ...
         break;
  default:
         program statement
         program statement
         ...
        break;
  }
```

The *expression* enclosed within parentheses is successively compared against the values *value1*, *value2, ..., valuen*, which must be simple constants or constant expressions. If a case is found whose value is equal to the value of *expression*, the program statements that follow the case are executed. Note that when more than one such program statement is included, they do *not* have to be enclosed within braces.

The break statement signals the end of a particular case and causes execution of the switch statement to be terminated. Remember to include the break statement at the end of every case. Forgetting to do so for a particular case causes program execution to continue into the next case whenever that case gets executed.

The special optional case called default is executed if the value of *expression* does not match any of the case values.

Example 4.2.6: Revising the Program to Evaluate Simple Expressions

```
/* Program to evaluate simple expressions of the form value operator value */

#include <stdio.h>
int main (void)
{
     float value1, value2;
     char operator;
     printf ("Type in your expression.\n");
     scanf ("%f %c %f", &value1, &operator, &value2);
     switch (operator)
     {
        case '+':
                    printf ("%.2f\n", value1 + value2);
                    break;
        case '-':
                   printf ("%.2f\n", value1 - value2);
                   break;
        case '*':
                   printf ("%.2f\n", value1 * value2);
                   break;
        case '/':
```

```
                if ( value2 == 0 )
                    printf ("Division by zero.\n");
                else
                    printf ("%.2f\n", value1 / value2);
                break;
        default:
                printf ("Unknown operator.\n");
                break;
    }
return 0;
}
```

Output

Type in your expression.
178.99 - 326.8
-147.81

4.3 ITERATION AND REPETITIVE EXECUTION

One of the fundamental properties of a computer is its ability to repetitively execute a set of statements.These *looping* capabilities enable you to develop concise programs containing repetitive processes that could otherwise require thousands or even millions of program statements to perform. The looping statement continues *as long as* the condition is satisfied. The *loop_condition* of the for statement is specified by any *relational expression.*

Table

Relational Operators	Operator Meaning	Example
==	Equal to	count == 10
!=	Not equal to	flag != DONE
<	Less than	a < b
<=	Less than or equal to	low <= high
>	Greater than	pointer > end_of_list
>=	Greater than or equal to	j >= 0

The relational operators have lower precedence than all arithmetic operators. This means, for example, that the following expression
a < b + c
is evaluated as
a < (b + c)
It would be TRUE if the value of a were less than the value of b +c and FALSE otherwise.
Pay particular attention to the “is equal to” operator == and do not confuse its use with the assignment operator =.The expression
a == 2
tests if the value of a is equal to 2, whereas the expression
a = 2
assigns the value 2 to the variable a.
The choice of which relational operator to use obviously depends on the particular test being made and in some instances on your particular preferences.
The C programming language contains three different program statements for program looping. They are known as

- the for statement,
- the while statement, and
- the do statement.

4.3.1 The for Statement

The general format of the for statement is as follows:

```
for ( init_expression; loop_condition; loop_expression )
    program statement
```

The three expressions that are enclosed within the parentheses—*init_expression, loop_condition*, and *loop_expression*—set up the environment for the program loop.
The program statement that immediately follows can be any valid C program statement and constitutes

the body of the loop. This statement is executed as many times as specified by the parameters set up in the for statement.

The first component of the for statement, labeled *init_expression*, is used to set the initial values *before* the loop begins.

The second component of the for statement the condition or conditions that are necessary *for* the loop to continue. In other words,
When the *loop_condition* is no longer satisfied, execution of the program continues with the program statement immediately following the for loop.

The final component of the for statement contains an expression that is evaluated each time *after* the body of the loop is executed. This expression is generally used to change the value of the index variable, frequently by adding 1 to it or subtracting 1 from it.
Remember that the looping condition is evaluated immediately on entry into the loop, before the body of the loop has even executed one time.Also, remember not to put a semicolon after the close parenthesis at the end of the loop (this immediately ends the loop).

Example 4.3.1

```
// Program to generate a table of triangular numbers
#include <stdio.h>
int main (void)
{
int  n, triangularNumber;
printf ("TABLE OF TRIANGULAR NUMBERS\n\n");
printf (" n    Sum from 1 to n\n");
printf ("---   ----------\n");
triangularNumber = 0;

for ( n = 1;  n <= 10;  n++ ) {
triangularNumber +=  n;
printf (" %i          %i\n", n, triangularNumber);
}
return 0;
}
```

Output

```
TABLE OF TRIANGULAR NUMBERS
n    Sum from 1 to n
---   ----------
1          1
2          3
3          6
4          10
5          15
6          21
7          28
8          36
9          45
10          55
```

4.3.2 Various forms of *for loop*

1) Here instead of n++, we can also use n=n+1

```
for ( n = 1;  n <= 10;  n=n+1)
```

2) Initialization part can be skipped from loop as shown below, the counter variable is declared before the loop itself.

```
int n=1;
for ( ; n<10 ; n++)
```

Must Note: Although we can skip init part but semicolon (;) before condition is must, without which you will get compilation error.

3) Like initialization, you can also skip the increment part. In this case semicolon (;) is must, after condition logic.

```
for (n=1;  n < =10; )
{
    //Code
    n++;
}
```

4) Here, neither the initialisation, nor the incrementation is done in the **for** statement, but still the two semicolons are necessary.

```
 int n=1;
for (;n<=10;)
{
    //Statements
    n++;
}
```

5) Counter can be decremented also, In the below example the variable gets decremented each time the loop runs until the condition num>10 becomes false.

```
for(num=20; num>10; num--)
```

6) It is also possible to declare a for loop without defining the initialisation part, condition part or increment part. Such loop will be executed infinite times, hence proper condition need to be mentioned inside the body of the loop to come out from the loop.

```
for ( ; ;)
{
//statements;
}
```

7) The initialisation expression of the **for** loop can contain more than one statement separated by a comma. For example,

```
for ( i = 1, j = 2 ; j <= 10 ; j++ )
```

Multiple statements can also be used in the incrementation expression of **for** loop; i.e., you can increment (or decrement) two or more variables at the same time. However, only one expression is allowed in the test expression. This expression may contain
several conditions linked together using logical operators.

4.3.3 While Loop

A while loop in C programming repeatedly executes a target statement as long as a given condition is true.

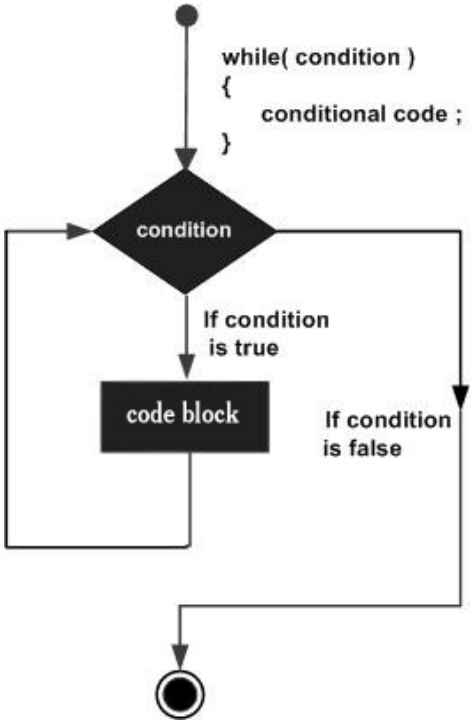
Syntax

The syntax of a while loop in C programming language is −

```
while(condition) {
   statement(s);
}
```

Here, statement(s) may be a single statement or a block of statements. The condition may be any expression, and true is any nonzero value. The loop iterates while the condition is true.
When the condition becomes false, the program control passes to the line immediately following the loop.

Flow Diagram



Here, the key point to note is that a while loop might not execute at all. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example 4.3.3

```
#include <stdio.h>

int main () {

  /* local variable definition */
  int a = 10;

  /* while loop execution */
  while( a < 20 ) {
    printf("value of a: %d\n", a);
    a++;
  }

  return 0;
}
```

Output:

When the above code is compiled and executed, it produces the following result −

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

4.3.4 Do … While Loop

Unlike **for** and **while** loops, which test the loop condition at the top of the loop, the **do...while** loop in C programming checks its condition at the bottom of the loop.

A **do...while** loop is similar to a while loop, except the fact that it is guaranteed to execute at least one time.

Syntax

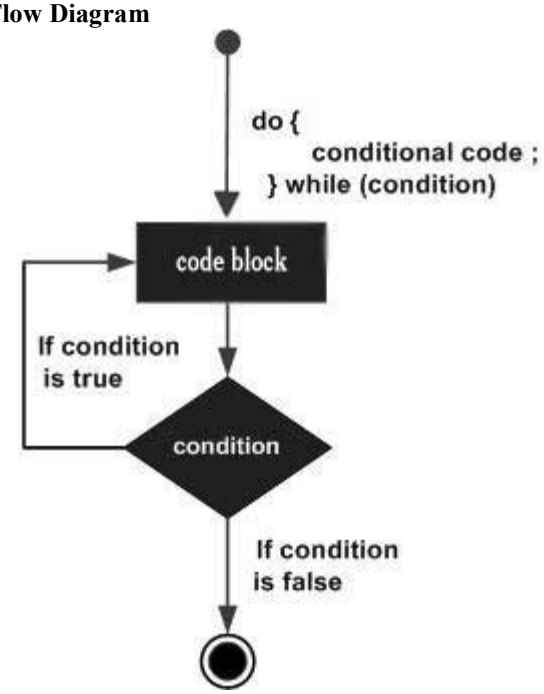
The syntax of a **do...while** loop in C programming language is −

```
do {
   statement(s);
} while( condition );
```

Notice that the conditional expression appears at the end of the loop, so the statement(s) in the loop executes once before the condition is tested.

If the condition is true, the flow of control jumps back up to do, and the statement(s) in the loop executes again. This process repeats until the given condition becomes false.

Flow Diagram



Example 4.3.4

```
#include <stdio.h>

int main () {

   /* local variable definition */
   int a = 10;

   /* do loop execution */
   do {
      printf("value of a: %d\n", a);
      a = a + 1;
   }while( a < 20 );

   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

4.3.5 Nested Loops

C programming allows to use one loop inside another loop. The following section shows a few examples to illustrate the concept.

Syntax

The syntax for a **nested for loop** statement in C is as follows −

```
for ( init; condition; increment ) {

   for ( init; condition; increment ) {
      statement(s);
   }

   statement(s);
}
```

The syntax for a **nested while loop** statement in C programming language is as follows −

```
while(condition) {

   while(condition) {
      statement(s);
   }

   statement(s);
}
```

The syntax for a **nested do...while loop** statement in C programming language is as follows −

```
do {

   statement(s);

   do {
      statement(s);
   }while( condition );

}while( condition );
```

A final note on loop nesting is that you can put any type of loop inside any other type of loop. For example, a 'for' loop can be inside a 'while' loop or vice versa.

Example 4.3.5.1

The following program uses a nested for loop to find the prime numbers from 2 to 100 −

```
#include <stdio.h>

int main () {

   /* local variable definition */
   int i, j;

   for(i = 2; i<100; i++) {

      for(j = 2; j <= (i/j); j++)
      if(!(i%j)) break; // if factor found, not prime
      if(j > (i/j)) printf("%d is prime", i);
   }

   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
2 is prime
3 is prime
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
23 is prime
```

29 is prime
31 is prime
37 is prime
41 is prime
43 is prime
47 is prime
53 is prime
59 is prime
61 is prime
67 is prime
71 is prime
73 is prime
79 is prime
83 is prime
89 is prime
97 is prime

Example 4.3.5.2: Program to print half pyramid using numbers

1
1 2
1 2 3
1 2 3 4
1 2 3 4 5

Source Code

```
#include <stdio.h>
int main()
{
   int i, j, rows;

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

   for(i=1; i<=rows; ++i)
   {
      for(j=1; j<=i; ++j)
      {
         printf("%d ",j);
      }
      printf("\n");
   }
   return 0;
}
```

4.4 goto Statement

A **goto** statement in C programming provides an unconditional jump from the 'goto' to a labeled statement in the same function.

NOTE − Use of **goto** statement is highly discouraged in any programming language because it makes difficult to trace the control flow of a program, making the program hard to understand and hard to modify. Any program that uses a goto can be rewritten to avoid them.

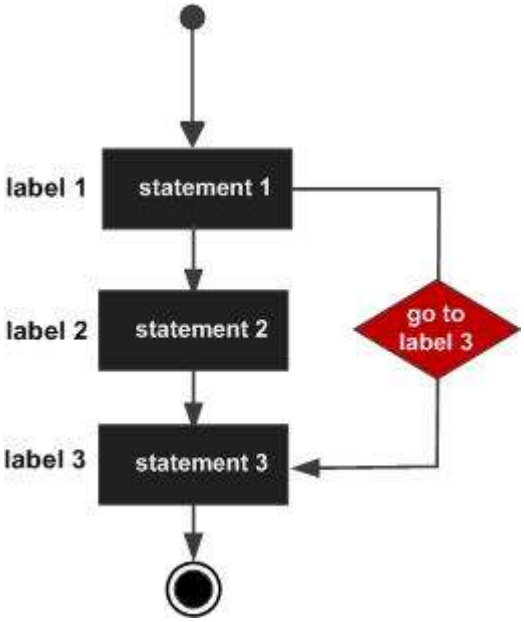
Syntax

The syntax for a **goto** statement in C is as follows −

```
goto label;
..
.
label: statement;
```

Here **label** can be any plain text except C keyword and it can be set anywhere in the C program above or below to **goto** statement.

Flow Diagram



Example 4.4

```
#include <stdio.h>

int main () {

  /* local variable definition */
  int a = 10;

  /* do loop execution */
  LOOP:do {

    if( a == 15) {
      /* skip the iteration */
      a = a + 1;
      goto LOOP;
    }

    printf("value of a: %d\n", a);
    a++;

  }while( a < 20 );

  return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

4.5 The *break* Statement

We often come across situations where we want to jump out of a loop instantly, without waiting to get back to the conditional test.The keyword **break** allows us to do this. When **break** is encountered inside any loop, control automatically passes to the
first statement after the loop. A **break** is usually associated with an **if**. As an example, let's consider the following example.

Example 4.5 : Write a program to determine whether a number is prime or not.

All we have to do to test whether a number is prime or not, is to divide it successively by all numbers from 2 to one less than itself. If remainder of any of these divisions is zero, the number is not a prime. If no division yields a zero then the number is a prime number. Following program implements this logic.

```
main( )
{
int num, i ;
printf ( "Enter a number " ) ;
scanf ( "%d", &num ) ;
i = 2 ;
while ( i <= num - 1 )
{
    if ( num % i == 0 )
    {
        printf ( "Not a prime number" ) ;
        break ;
    }
    i++ ;
}
if ( i == num )
    printf ( "Prime number" ) ;
}
```

In this program the moment **num % i** turns out to be zero, (i.e. **num** is exactly divisible by **i**) the message "Not a prime number" is printed and the control breaks out of the **while** loop.

The keyword **break**, breaks the control only from the **while** in which it is placed. Consider the following program, which illustrates this fact.

```
main( )
{
   int i = 1 , j = 1 ;
   while ( i++ <= 100 )
   {
       while ( j++ <= 200 )
       {
           if ( j == 150 )
               break ;
           else
               printf ( "%d %d\n", i, j ) ;
       }
   }
}
```

In this program when **j** equals 150, **break** takes the control outside the inner **while** only, since it is placed inside the inner **while**.

4.6 The *continue* Statement

In some programming situations we want to take the control to the beginning of the loop, bypassing the statements inside the loop, which have not yet been executed. The keyword **continue** allows us to do this. When **continue** is encountered inside any loop, control automatically passes to the beginning of the loop.

A **continue** is usually associated with an **if**. As an example, let's consider the following program.

Example 4.6

```
main( )
{
    int i, j ;
    for ( i = 1 ; i <= 2 ; i++ )
    {
        for ( j = 1 ; j <= 2 ; j++ )
        {
            if ( i == j )
                continue ;
```

```
                printf ( "\n%d %d\n", i, j ) ;
            }
        }
    }
```

The output of the above program would be...

1 2

2 1

Note that when the value of **i** equals that of **j**, the **continue** statement takes the control to the **for** loop (inner) bypassing rest of the statements pending execution in the **for** loop (inner).

4.7 Summary

- A program is usually not limited to a linear sequence of instructions. During its process it may require to repeat execution of a part of code more than once deepening upon the requirements or take decisions. For that purpose, C provides control and looping statements. In this unit, we had seen the different looping statements provide by C language namely while, do... while and for.
- Using break statement, we can leave a loop even if the condition for its end is not fulfilled. It can be used to end an infinite loop, or to force it to end before its nature end. The continue statement causes the program to skip the rest of the loop in the present iteration as if the end of the statement block would have reached, causing jump to the following iteration.
- Using the go to statement, we can make an absolute jump to another point in the program. You should use this feature carefully since its execution ignores any type nesting limitation. The destination point is identified by a label, which is then used argument for the go to instruction. A label is made of a valid identifier followed by a colon (:).

4.8 Questions for Exercise

1. Write a program to generate and display a table of n and n^2, for integer values of n ranging from 1 to 10. Be certain to print appropriate column headings.

2. A triangular number can also be generated by the formula
   triangularNumber = n (n + 1) / 2, for any integer value of n.
   For example, the 10th triangular number, 55, can be generated by substituting 10 as the value for n in the preceding formula.Write a program that generates a table of triangular numbers using the preceding formula.
   Have the program generate every fifth triangular number between 5 and 50 (that is, 5, 10, 15, ..., 50).

3. The factorial of an integer n, written n!, is the product of the consecutive integers 1 through n. For example, 5 factorial is calculated as 5! = 5 x 4 x 3 x 2 x 1 = 120. Write a program to generate and print a table of the first 10 factorials.

4. Write a program that calculates the sum of the digits of an integer. For example, the sum of the digits of the number 2155 is 2 + 1 + 5 + 5 or 13.The program should accept any arbitrary integer typed in by the user.

5. Write a program that asks the user to type in two integer values at the terminal. Test these two numbers to determine if the first is evenly divisible by the second, and then display an appropriate message at the terminal.

6. Write a program that accepts two integer values typed in by the user. Display the result of dividing the first integer by the second, to three-decimal-place accuracy. Remember to have the program check for division by zero.

7. Write short notes on go to statement?

8. Mention difference between While loop and do…While loop. What is Nested Loop?

9. Explain various conditional control structures in C.

10. Explain various conditional looping statements in C.

11. Write the differences between Break and Continue.

12. Write a menu driven program which has following options:
    1. Factorial of a number.
    2. Prime or not
    3. Odd or even
    4. Exit

4.9 Suggested Readings

1. The C programming language, Brain W Kernighan, Dennis M Ritchie, PHI.
2. Programming with C, Second Edition, Byron Gottfried, Tata McGraw Hill, 2003.
3. C. The Complete Reference, Fourth Edition, Herbert Schildt, Tata McGraw Hill,
4. 2002.
5. Computer Science: A Structured Programming Approach Using C, Second Edition, Behrouz A. Forouzan, Richard F. Gilberg, Brooks/Cole Thomas Learning, 2001.
6. The C Primer, Leslie Hancock, Morris Krieger, Mc Gravy Hill, 1983.