4. FLOW CONTROL STATEMENTS

We've been concentrating on constants and variables, and the operators with which we combine them into expressions. Now we turn our attention to C's rather small set of statements. For this we present the fine points of the statements we have already used, and describe in detail those we have so far overlooked. While doing so, we produce a collection of useful functions for manipulating arrays of integers, and we write programs to simulate a simple calculator, and to print various breakdowns of their input characters.

4.1 Expression and Compound statements

C's simplest statement is the ***expression statement***, an expression followed by a semicolon. We can use an expression statement anywhere C's syntax requires a statement. Both assignment statements and function calls, such as

```
sum += value ;
```

and

```
printf (" Sum is = %d \n" , sum ) ;
```

are merely expression statements and not special statement types, as in FORTRAN or BASIC.

An expression statement executes by evaluating the expression and discarding its result. That means that to be useful, an expression statement must have a side effect, such as invoking a function or changing a variable's value. Unfortunately, it's easy to accidentally write a syntactically legal expression statement that accomplishes nothing. The legal but useless expression statement

```
sum / entries ;
```

divides *sum* by *entries* but does nothing with the result. Luckily, Turbo C often warns you about such statements.

A ***compound statement*** or ***block*** is a group of statements surrounded by braces. Like expression statements, we can place compound statements anywhere C's syntax requires a statement. But unlike expression statements, we don't follow them with a semicolon.

When we have a compound statement composed entirely of expression statements, we can use the comma operator to turn it into a single, more compact expression statement. For example, we can replace

```
{  temp = x  ;  x = y  ;  y = temp  ;  }
```

with

```
temp = x  ,  x = y  ,  y = temp  ;
```

Transformations like this one lead to more compact but less readable programs. They also make our programs harder to modify and debug.

4.2 Simple Decisions

if statement

The *if* statement is used to execute a segment of code conditionally. The simplest form of the *if* statement is

```
if ( expression )

    action ;
```

Notice that the **expression** must be enclosed in parentheses. To execute an *if* statement, the expression must be evaluated to either TRUE (any non-zero value) or FALSE (0). If ***expression*** is TRUE, the *action* will be performed and execution will continue on to the next statement following the *action*. However, if ***expression*** evaluates to FALSE, the *action* will not be executed, and the statement following *action* will be executed. For example, the following code segment will print the message " ***Not a quadratic equation*** " whenever the variable *a* equals 0 in a quadratic equation :

```
if ( a == 0 )

        printf (" Not a quadratic equation ") ;
```

The syntax for an *if* statement associated with a block of executable statements looks like this :

```
if ( expression )
{
        action ;

        action ;

        action ;
}
```

The syntax requires that all of the associated statements be enclosed in a pair of braces **{ }** and that each statement within the block end with a semicolon. Here in an example compound *if* statement :

```
if ( a == 0 )
{
        printf (" Not a quadratic equation ") ;

        exit( ) ;
}
```

Eg : Write a C program to calculate the absolute value of an integer.

```
# include <stdio.h>

void main (void)
{
        int number ;

        printf (" Type in your number ") ;
        scanf ("%d" , &number);

        if ( number < 0 )
                number = -number ;

        printf (" The absolute value is %d\n " , number ) ;
}
```

4.3 Multiway Decisions

if-else statement

The ***if-else*** statement allows a program to take two separate actions based on the validity of a particular expression. The simplest syntax for an ***if-else*** statement looks like this :

```
if ( expression )

        action1 ;

else

        action2 ;
```

In this case, if ***expression*** evaluates to TRUE, *action1* will be taken. Otherwise, when ***expression*** evaluates to FALSE, *action2* will be executed. A coded example looks like this :

```
if ( a > b )

        max = a ;

else

        max = b ;
```

Of course, either *action1*, *action2*, or both could be compound statements, or blocks, requiring braces. The syntax for these three combinations is straightforward :

```
if ( expression )
{
        action1a ;

        action1b ;

        action1c ;
}
else
        action2 ;
```

```
if ( expression )
        action1 ;
else
{
        action2a ;

        action2b ;

        action2c ;
}
```

```
if ( expression )
{
        action1a ;

        action1b ;

        action1c ;
}
else
{
        action2a ;
        action2b ;

        action2c ;

}
```

Remember, whenever a block *action* is being taken, you don't follow the closing brace with a semicolon.

The following C program segment uses an ***if-else*** statement twice; both *if* and *else* is a compound block :

```
if ( a == 0 )
{

        printf (" Not a quadratic equation ") ;

        exit( ) ;
```

```
}
else
{
        d = b*b - 4*a*c ;

        if ( d < 0 )
        {

                printf (" Complex Roots ") ;

                exit( ) ;
        }
        else
        {

                Root1 = ( -b + sqrt( d ) ) / ( 2 * a ) ;

                Root2 = ( -b - sqrt( d ) ) / ( 2 * a ) ;

        }

}
```

Eg : Write a C program to determine if a number is even or odd.

```
# include <stdio.h>

void main (void)
{
        int number_to_test , remainder ;

        printf (" Enter your number to be tested : \n " ) ;
        scanf ("%d" , &number_to_test );

        remainder = number_to_test % 2 ;

        if ( remainder == 0 )
                printf (" The number is even \n " ) ;
        else
                printf (" The number is odd \n " ) ;
        }
```

Ex : Write a C program to find the largest of three given numbers.

(a) Using *if-else*
(b) Using *Conditional Operator*

Ex : Write a C program to solve a quadratic equation of the form $ax^2 + bx + c = 0$. Roots of the quadratic equation are : $(- b + (b^2 - 4ac)^{1/2}) / (2 * a)$ and
$(- b - (b^2 - 4ac)^{1/2}) / (2 * a)$

Nested *if-elses*

When nesting ***if*** statements, make sure that you know which ***else action*** will be matched up with which *if*. See if you can determine what will happen in this example :

```
if ( Temperature < 50 )
if ( Temperature < 30 )

        printf (" "Wear the down jacket1 ") ;

else

        printf (" Parka will do ") ;
```

The listing is purposely misaligned so you have no visual clues about which statement goes with which *if*. If Temperature is 55, does the "*Parka will do*" message get printed ? The answer is no. In this example, the *else action* is associated with the second *if expression*.

Of course, proper indentation will always help clarify the situation :

```
if ( Temperature < 50 )
        if ( Temperature < 30 )

                printf (" "Wear the down jacket1 ") ;

        else

                printf (" Parka will do ") ;
```

Each application you write will benefit most by the styles, as long as you are consistent throughout the source code.

See if you can figure out this example :

```
if ( Temperature < 50 )
        if ( Temperature < 30 )

                printf (" "Wear the down jacket1 ") ;

else

        printf (" Parka will do ") ;
```

This looks like just another example of what has been discussed. However, what if you wanted " *Parka will do* " to be associated with expression " *Temperature < 50* " rather than " *Temperature < 30* " ? The examples so far have all associated the *else action* with the second or closest *if*. They're indented to work the way you are logically thinking (as was the previous example); unfortunately, the compiler disregards indentation.

To correct this situation, you need to use braces :

```
if ( Temperature < 50 )
{
        if ( Temperature < 30 )

                printf (" "Wear the down jacket1 ") ;
}

else

        printf (" Parka will do ") ;
```

You solve the problem by making " *Temperature < 30* " and its associated " *Wear the down jacket1* " a block associated with a TRUE evaluation of the "*Temperature<50* ". This makes it clear that " *Parka will do* " will be associated with the else clause of " *Temparature < 50* ".

if-else-if statements

The ***if-else-if*** statement is often used to perform multiple successive comparisons. Its general form looks like this :

```
if ( expression1 )

        action1 ;

else if ( expression2 )

        action2 ;

else if ( expression3 )

        action3 ;
```

Of course, each action could be a compound block requiring its own set of braces (with the closing brace not followed by a semicolon). This type of logical control flow evaluates each expression until it finds one that is TRUE. When this occurs, all remaining test conditions are bypassed. In the previous example, no action would be taken if none of the expressions evaluated to TRUE.

See if you can guess the result of this next example.

```
if ( expression1 )
        action1 ;
else if ( expression2 )
        action2 ;
else if ( expression3 )
        action3 ;
else
        default_action ;
```

Unlike the previous example, this *if-else-if* statement will always perform some *action*. If none of the ***if(expression)***s evaluates to TRUE, the ***else default_action*** will be executed. For example, the following program segment checks the value assigned to *Average* to decide which message to print. If the Average is not one of the limits provided, the code segment prints an appropriate message.

```
if ( ( Average <= 100 ) && ( Average >= 70 )
        printf (" Very Good \n ") ;
else if ( ( Average < 70 ) && ( Average >= 55 )
        printf (" Good \n ") ;
else if ( ( Average < 55 ) && ( Average >= 40 )
        printf (" Pass \n ") ;
else if ( ( Average < 40 ) && ( Average >= 0 )
        printf (" Fail \n ") ;
else
        printf (" Invalid Marks \n") ;
```

Ex : The results of an examination is issued after calculating the average of the marks of 3 subjects. The grades of the examination results are categorized into four groups as follows :

Average >= 70	-	Very Good
70 > Average >= 55	-	Good
55> Average >= 40	-	Pass
Average < 40	-	Fail

Write a C program to decode the grade of a student when his/her subject marks are given. The error message " ***Invalid Marks*** " should output when negative average is computed (if any).

(a) Using *if-else-if*
(b) Using *Conditional Operator*

Ex : Write a C program to convert Centigrade into Fahrenheit and Fahrenheit into Centigrade.

Eg : Write a C program to simulate a pocket calculator which can be able to perform normal calculations such as *Addition*, *Subtraction*, *Multiplication*, and *Division*.

```
# include < stdio.h >

void main ( void )
{
        float Value1, Value2 ;
        char Operator ;

printf ("Enter your expression of the form value  operator  value \n") ;
scanf ("%f%c%f" , &Value1, &Operator, &Value2 ) ;

if ( Operator == '+' )
        printf ("%f \n" , Value1 + Value2 ) ;

else if ( Operator == '-' )
        printf ("%f \n" , Value1 - Value2 ) ;

else if ( Operator == '*' )
        printf ("%f \n" , Value1 * Value2 ) ;

else if ( Operator == '/' )
        if ( Value2 != 0 )
                printf ("%f \n" , Value1 / Value2 ) ;
        else
                printf (" Warning : Division by Zero \n " ) ;

else
```

```
        printf (" Unknown Operator \n " ) ;
}
```

The next program analyzes a character that is typed in from the terminal and classifies it as either an alphabetic character (a - z or A - Z), a digit (0 - 9), or a special character (anything else). In order to read a single character from the terminal, the format character %c are used in the *scanf* call.

/* This program categorizes a single character that is entered at the terminal ***/**

```
void main (void)
{
        char C ;

        printf (" Enter a single character : \n ") ;
        scanf ("%c" , &C ) ;

        if ( ( C >= 'a' && C<= 'z' ) || ( C >= 'A' && C<= 'Z' ) )
                printf (" It's an alphabetic character \n ") ;

        else if ( C >= '0' && C<= '9' )
                printf (" It's a digit \n ") ;

        else
                printf (" It's a special character \n ") ;
}
```

The break statement

The C **break** statement can be used to exit a loop before the test condition becomes FALSE. The **break** statement is similar in many ways to a **goto** statement, only the point jumped to is not known directly. When breaking out of a loop, program execution continues with the next statement following the loop itself.

```
void main (void)
{
        int i = 1 , Sum = 0 ;

        while ( i < 10 )
        {
                Sum += i ;

                if ( Sum > 20 )

                        break ;

                        i++ ;
```

```
        }
}
```

Use the debugger to trace through the program. Trace the variables *Sum* and *I*. Pay particular attention to which statements are executed after *Sum* reaches the value 21. Notice that when *Sum* reaches the value 21, the **break** statement is executed. This causes the increment of *i* to be jumped over, *i*++, with program execution continuing on the line of code below the loop.

The continue statement

There is a subtle difference between the C **break** statement and the C **continue** statement. As you have seen, **break** causes the loop to terminate execution altogether. In contrast, **continue** causes all of the statements following it to be ignored but does not circumvent incrementing the loop control variable or the loop control test condition. In other words, if the loop control variable still satisfies the loop test condition, the loop will continue to iterate.

Using break and continue statements Together

Both the break and continue statements can be combined to solve some interesting program problems. Consider the following example :

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

void main (void)
{
        int C ;

        while ( ( C = getchar( ) ) != EOF )
        {
                if( (C >= ‘a’ && C<= ‘z’ ) || (C >= ‘A’ && C<= ‘Z’) )
                {
                        printf (“ It’s an alphabetic character  \n ”) ;

                        printf (“ Going to continue \n ”) ;

                        continue ;
                }

                else
                {
                        printf (“ Not going to continue \n ”) ;

                        break ;
                }
```

```
        }
}
```

The goto statement

The **goto** is the final control-flow altering command. It simply transfers control to a labeled statement.

goto *label*

.....

label : statement

A label has the same syntax as an identifier and must be in the same block as the **goto**. A null statement must follow any label located at the end of a function. One reasonable use of a **goto** is to rapidly bail out of nested loops when an error occurs.

The exit statement

Under certain circumstances, it is proper for a program to terminate long before all of its statements have been examined and/or executed. For these circumstances, C incorporates the **exit** library function. The **exit** function expects one integer argument called a *status value*. The UNIX and MS-DOS operating systems interpret a status value of 0 as a normal program termination and any nonzero status values as different kinds of errors.

The process that invoked the program can use the particular status value passed to **exit** to take some action. For example, if the program were invoked from the command line and the status value indicated some type of error, the operating system might display a message. In addition to terminating the program, **exit** writes all output waiting to be written and closes all open files. Either *process.h* or *stdlib.h* can be included to prototype the function **exit**. Including the *stdlib.h* header file instead of *process.h* makes visible two additional definitions. **EXIT_SUCCESS** (returns a value of 0) and **EXIT_FAILURE** (returns an unsuccessful value).

Eg : exit (**EXIT_FAILURE**) ;

switch statement

You will often want to test a variable or an expression against several values. You could use nested ***if-else-if*** statements to do this, or you could use a ***switch*** statement. Be very careful. Unlike many other high-level language selection statements, the C ***switch*** statement has a few peculiarities. The syntax for a ***switch*** statement looks like this :

```
switch ( integral expression )
        {
                case constant1   :      statement1 ;  break ;

                case constant2   :      statement2 ;  break ;
                        .
                        .
                        .
                        .
                        .
                case constantn   :      statementn ;  break ;

                default          :      statement ;
        }
```

The **break** statement causes an immediate exit from the ***switch***. Because cases serve just as labels, after the code for one case is done, execution *falls through* to the next unless you take explicit action to escape. **break** and **return** are the most common ways to leave a *switch*. A **break** statement can also be used to force an immediate exit from *while*, *for*, and *do-while* loops, as will be discussed later in this chapter.

Falling through cases is a mixed blessing. On the positive side, it allows several cases to be attached to a single action, as with the digits in this example. But it also implies that normally each case must end with a **break** to prevent falling through to the next. Falling through from one case to another is not robust, being prone to disintegration when the program is modified. With the exception of multiple labels for a single computation, fall through should be used sparingly, and commented.

As a matter of good form, put a **break** after the last case (the default here) even though it's logically unnecessary. Some day when another case gets added at the end, this bit of defensive programming will save you.

Pay particular attention to the **break** statement. If this example had been coded in Pascal and *constant1* equaled the *integral expression*, *statement1* would have been executed, with program execution picking up with the next statement at the end of the *switch* statement (below the closing brace).

In C, the situation is quite different. If the **break** statement had been removed from *constant1's* code segment, a similar match used in the previous paragraph would have left *statement2* the next statement to be executed. The **break** statement causes the remaining portion of the *switch* statement to be skipped.

The following ***if-else-if*** code segment

```
if ( x == 4 )
        y = 7 ;

else if ( x == 5 )
```

```
        y = 9 ;

else if ( x == 9 )
        y = 14 ;

else
        y = 22 ;
```

can be rewritten using a ***switch*** statement :

```
switch ( x )
{
        case  4  :  y = 7 ; break ;

        case  5  :  y = 9 ; break ;

        case  9  :  y = 14 ; break ;

        default  :  y = 22 ; break ;
}
```

In this example, the value of *x* is consecutively compared to each *case* value looking for a match. When one is found, *y* is assigned the appropriate value and then the **break** statement is executed, skipping over the remainder of the *switch* statements. However, if no match is found, the ***default*** assignment is performed (y == 22). Since this is the last option in the ***switch*** statement, there is no need to include a **break**. A ***switch default*** is optional.

Eg :

```
# include <stdio.h>

void main (void)
{
        char ch ;
        int Counta = 0, Countb = 0, Countc = 0 ;

        printf ( " Enter the character \n " ) ;
        scanf ("%c" , &ch ) ;

        switch ( ch )
        {
                case  'a'  :  Counta++ ; break ;

                case  'b'  :  Countb++ ; break ;

                case  'c'  :  Countc++ ; break ;

                default    :  printf ( " Not a, b , or c \n " ) ;
        }
```

```
        printf ( " %3d%3d%3d\n " , Counta , Countb , Countc );

}
```

Ex : Characters from A to Z are grouped as follows :

```
Group 0  -   A , E , I , O , U
Group 1  -   B , D , Q , P , R
Group 2  -   C , S , G
Group 3  -   M , N , K , L , F , H , X
Group 4  -   J , T
Group 5  -   V , W
Group 6  -   Y , Z
```

Write a C program using a ***switch*** statement to find in which group a character is when the character is given as input data. Assume for simplicity that lowercase letters do not occur.

Eg : Write a C program to simulate a pocket calculator which can be able to perform normal calculations such as *Addition*, *Subtraction*, *Multiplication*, and *Division* using *switch* statement..

```
# include < stdio.h >

void main ( void )
{
        float Value1, Value2 ;
        char Operator ;

printf ("Enter your expression of the form value  operator  value \n" );
scanf ("%f%c%f" , &Value1, &Operator, &Value2 ) ;

        switch ( Operator )
        {
                case  '+' :  printf ("%f \n" , Value1 + Value2 ) ; break;

                case  '-' :  printf ("%f \n" , Value1 - Value2 ) ; break ;

                case  '*' :  printf ("%f \n" , Value1 * Value2 ) ; break ;

                case  '/' :  if ( Value2 != 0 )
                                printf("%f \n",Value1 / Value2 ) ;
                             else
                                printf("Warning : Division by Zero\n");
                                break ;

                default   :  printf (" Unknown Operator \n " ) ; break ;
```

```
        }

}
```

Eg : **/*** A C program demonstrating the switch statement ***/**

```
# include < stdio.h >

double Add ( float , float ) ;
double Sub ( float , float ) ;
double Mul ( float , float ) ;
double Div ( float , float ) ;
void Error (void ) ;

void main ( void )
{
        float Value1, Value2 ;
        char Operator ;

printf ("Enter your expression of the form value  operator  value \n" );
scanf ("%f%c%f" , &Value1, &Operator, &Value2 ) ;

        switch ( Operator )
        {
                case  '+' :  printf ("%lf \n" , Add(Value1,Value2) ) ;
                                break ;

                case '-' : printf ("%lf \n" , Sub(Value1,Value2) ) ;
                        break ;

                case '*' :  printf ("%lf \n" , Mul(Value1,Value2) ) ;
                        break ;

                case  '/' :  if ( Value2 != 0 )
                                printf ("%lf \n" , Div(Value1,Value2 ) );
                             else
                                Error ( ) ; break ;

                default   :  printf (" Unknown Operator \n " ) ; break ;
        }

}


double Add ( float Value1 , float Value2 )
{
        return ( Value1 + Value2 ) ;
}
```

```
double Sub ( float Value1 , float Value2 )
{
        return ( Value1 - Value2 ) ;
}


double Mul ( float Value1 , float Value2 )
{
        return ( Value1 - Value2 ) ;
}


double Div ( float Value1 , float Value2 )
{
        return ( Value1 / Value2 ) ;
}


void Error ( void )
{
        printf ( " Warning : Division by Zero \n " ) ;
}
```

4.4 Loop statements

The C language includes the standard set of repetition control statements : *for* loops, ***while*** loops, and ***do-while*** loops (called ***repeat-until*** loops in several other high-level languages). However, C provides four methods for altering the repetitions in a loop. All repetition loops can naturally terminate based on the expressed Boolean test condition. In C, however, a repetition loop can also terminate because of an anticipated error condition using either a **break** or **exit** statement. Repetition loops can also have their logic control flow altered by **break** or **continue** statements.

The basic difference between a ***for*** loop and a *while* or ***do-while*** loop has to do with the known number of repetitions. Typically, *for* loops are used whenever there is a definite predefined required number of repetitions. In contrast, ***while*** and ***do-while*** loops are reserved for an unknown number of repetitions.

for loop

The syntax for a ***for*** loop looks like this :

```
for ( initialization ; condition ; increment )

        statement ;
```

When the ***for*** loop statement is encountered, the **initialization** is executed first, at the start of the loop, and is never executed again. Usually, this statement involves the initialization of the loop control variable. Following this, the ***condition*** which is called the loop terminating condition, is tested. Whenever the *condition* evaluates to TRUE, the statement or statements within the loop are executed. If the loop was entered, the *increment* is executed after all of the statements within the loop are executed. However, if *condition* evaluates to FALSE, the statements within the loop are ignored, along with the *increment*, and execution continues with the statement following the end of the loop. The indentation scheme for *for* loops with several statements to be repeated looks like this :

```
for ( initialization ; condition ; increment )
{
        statement_a ;

        statement_b ;

        statement_c ;

        statement_n ;
}
```

When several statements need to be executed, a pair of braces, **{ }**, is required to tie their execution to the loop control structure.

The following example sums up the first ten integers. It assumes that *i* and *Sum* have been predefined as integers.

```
Sum = 0 ;
for ( i = 0  ;  i <= 10  ;  i++ )

    Sum = + i ;
```

or

```
for ( Sum = 0 , i = 0  ;  i <= 10  ;  i++ )

    Sum = + i ;
```

or

```
for ( Sum = 0 , i = 0  ;  i <= 10  ;  Sum = + i , i++ ) ;
```

Grammatically, the three components of a ***for*** loop are expressions. Most commonly, $\boldsymbol{expr_1}$ and ***expr3*** are assignments or function calls and ***expr2*** is a relational expression. Any of the three parts can be omitted, although the semicolons must remain. If ***expr1*** or ***expr3*** is omitted, it is simply dropped from the expansion. If the ***condition, expr2***, is not present, it is taken as permanently true, so

```
for ( ; ; )              or          for (  ; i < MAX ;  )
{                                    {

        .....                                .....

}                                    }
```

is an "**infinite**" loop, presumably to be broken by other means, such as a **break**, **exit** or **return**.

Eg : Write a program to calculate the triangular number

```
# include < stdio.h >

void main ( void )
{
        int  n , number , triangular_number ;

        printf ( " What triangular number do you want : \n " ) ;
        scanf ( "%d" , &number ) ;

        triangular_number = 0 ;

        for ( n = 1 ; n <= number ; ++n )
                triangular_number += n ;

        printf ( " Triangular number %d is = %d \n " ,number , /
        triangular_number ) ;
}
```

***** A triangular number can also be generated by the formula : n (n + 1) / 2

Nested *for* loops

When you combine ***for*** loops, as in this next example, take care to include the appropriate braces **{ }**, to make certain the statements execute properly :

/* A C program demonstrating the need for caution when nesting for loops ***/**

```
# include < stdio.h >

void main( void )
{
        int outer_value , inner_value ;
```

```
        for ( outer_value = 1 ; outer_value <= 4 ; outer_value++ )
        {
                printf ( " \n %3d  --" , outer_value );

               for(inner_value = 1 ; inner_value<= 5 ; inner_value++)

                printf ( " %3d " , outer_value * inner_value ) ;
        }
}
```

The output produced by this program looks like this :

```
1  --  1   2   3   4   5
2  --  2   4   6   8  10
3  --  3   6   9  12  15
4  --  4   8  12  16  20
```

Had the outer *for* loop been written without the braces, like this

```
/*      A C program demonstrating the need for caution when nesting for loops     */

        # include < stdio.h >

        void main( void )
        {
                int outer_value , inner_value ;

                for ( outer_value = 1 ; outer_value <= 4 ; outer_value++ )

                        printf ( " \n %3d  --" , outer_value );

                       for(inner_value = 1 ; inner_value<= 5 ; inner_value++ )

                        printf ( " %3d " , outer_value * inner_value ) ;
        }
```

the output would have looked quite different :

```
1  --
2  --
3  --
4  --  5  10  15  20   25
```

while loop

Just like the ***for*** loop, the C ***while*** loop is a *pretest* loop. This means that ***condition*** is evaluated before the statements within the body of the loop are entered. Because of this, pretest loops may be executed from zero to many times. The syntax for a C ***while*** looks like this :

```
while ( condition )

        statement ;
```

For ***while*** loops with several statements, braces are needed :

```
while ( condition )
{
        statement_a ;

        statement_b ;

        statement_c ;

        statement_n ;
}
```

Usually ***while*** loop control structures are used whenever an indefinite number of repetitions is expected.

The ***for*** statement

```
for ( expr1  ;  expr2  ;  expr3 )

        statement ;
```

is equivalent to

```
expr1 ;

while ( expr2 )
{
        statement ;

        expr3 ;
}
```

Eg : Write a C program to reverse the digits of a number

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

```
void main ( void )
{
        int number , right_digit ;

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

        while ( number != 0 )
        {
                right_digit = number % 10 ;

                printf ( "%d" , right_digit ) ;

                number = number / 10 ;
        }
}
```

do-while loop

The do-while loop differs from both the ***for*** and ***while*** loops in that it is a *post-test* loop. In other words, the loop is always entered at least once, and the loop condition is tested at the end of the first iteration. In contrast, ***for*** and ***while*** loops may execute from zero to many times, depending on the loop control variable. Since ***do-while*** loops always execute at least one time, they are best used whenever you are certain that you want the particular loop entered. For example, your program may need to present a menu to the users even if they just want to immediately quit the program. They will need to see the menu to know which key terminates the application.

The syntax for a ***do-while*** loop looks like this :

```
do
        statement ;

while ( condition ) ;
```

Braces are required for ***do-while*** statements that have compound actions :

```
do
{
        statement_a ;

        statement_b ;

        statement_c ;

        statement_n ;
```

```
} while ( condition ) ;
```

Eg : Write a C program to reverse the digits of a number using *do-while* statement

```
# include < stdio.h >

void main ( void )
{
        int number , right_digit ;

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

        do
        {
                right_digit = number % 10 ;

                printf ( "%d" , right_digit ) ;

                number = number / 10 ;

        } while ( number != 0 ) ;

}
```

The following C program uses a ***do-while*** loop to print a menu and obtain a valid user response :

```
# include < stdio.h >

void main ( void )
{
        int  ch ;

        do
        {
                gotoxy( 5 , 2 ) ;   printf ( " MAIN MENU " ) ;
                gotoxy( 3 , 3 ) ;   printf ( " File " ) ;
                gotoxy( 3 , 4 ) ;   printf ( " Edit " ) ;
                gotoxy( 3 , 5 ) ;   printf ( " Search " ) ;
                gotoxy( 3 , 6 ) ;   printf ( " Compile " ) ;
                gotoxy( 3 , 7 ) ;   printf ( " Debug " ) ;
                gotoxy( 3 , 8 ) ;   printf ( " Project " ) ;
                gotoxy( 3 , 9 ) ;   printf ( " Options " ) ;
                gotoxy( 3 , 10 ) ;   printf ( " Help " ) ;
                gotoxy( 3 , 11 ) ;   printf ( " Quit " ) ;
```

```
            ch = toupper( getche( ) ) ;

        } while ( !strchr( "FESCDPOHQ" , ch ) ) ;

    }
```

Ex : Write a C program to find the sum of $1 + 2 + 3 + 4$ ..... $+ 19$

(a) Using the *for* statement

(b) Using the *while* statement

(c) Using the *do-while* statement

Modify the above program to find the sum of $1^3 + 2^3 + 3^3 + 4^3$ ..... $+ 19^3$

Ex : Write a C program to find the factorial of a given number

(a) Using the *for* statement

(b) Using the *while* statement

(c) Using the *do-while* statement

Ex : Write a C program to find how many times a man has to doubled 1 gram of rice to obtain more than 1 million gram of rice.

Flags

Just about anyone learning to program soon finds him or herself with the task of having to write a program to generate a table of ***prime numbers***. To refresh your memory, a positive integer *p* is a prime number if it not evenly divisible by any other integers, other than 1 and itself. The first prime integer is defined to be 2. The next prime is 3; and 4 is *not* prime because it is evenly divisible by 2.

There are several approaches that we could take in order to generate a table of prime numbers. If we had the task to generate all prime numbers up to 50, for example, then the most straightforward (and simplest) algorithm to generate such a table would simply test each integer *p* for divisibility by all integers from 2 through *p*-1. If any such integer evenly divided *p*, then *p* would not be prime; otherwise, it would be a prime number.

Eg : Write a C program to generate a table of prime numbers

```
void main (void)
{
        int  p ,  is_prime  ,  d  ;

        for ( p = 2 ; p <= 50 ; ++p )
        {

                is_prime = 1 ;

                for ( d = 2 ; d < p ; ++d )
                        if ( p % d == 0 )
                                is_prime = 0 ;

                if ( is_prime != 0 )
                        printf ( " %d " , p ) ;
        }
        printf ( " \n " ) ;
}
```

Eg : The next program counts lines, words, and characters, with the loose definition that a word is any sequence of characters that does not contain a blank, tab or newline.

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

void main(void)
{
        int ch, Flag =1, Ccount = 0, Wcount = 0, Lcount = 0;

        clrscr( );

        while ( ( ch = getchar( ) ) != EOF )
        {

                ++Ccount ;

                if ( ( ch != ' ' ) && ( ch != '\n' ) )
                {
                        if( Flag )
                        {
                                ++Wcount ;
                                Flag = 0 ;
                        }
                }
                else if ( ch == '\n' )
```

```
                {
                        ++Lcount ;
                        Flag = 1 ;
                }
                else
                        Flag = 1 ;
        }

        printf("Characters = %d , Words = %d , Lines = %d " , /
                Ccount, Wcount , Lcount ) ;
}
```