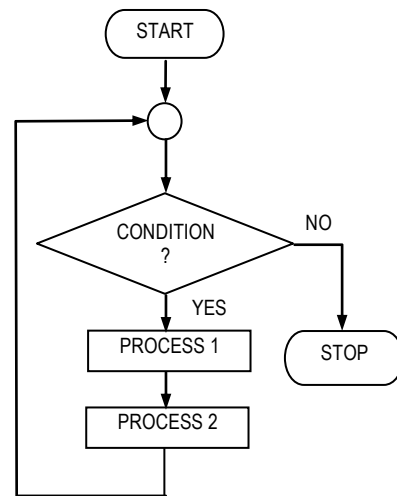


## C-04 Loop Structures in C

### 4.1 General

Apart from conditional branching there may be situations where a particular set of instructions may have to be repeated several number of times depending upon some conditions, until the desired result is achieved. Such a process is also called **iteration**. The number of times to repeat may be a fixed value or may be a variable one depending upon the satisfaction of certain conditions. The general logic for such a situation is given by the flowchart to the right. As long as the condition is true (Yes), process1 and process2 are repeated. Each time after the execution of the blocks, the condition is checked to be true or false. As soon as the condition is found to be false the program control comes out of the loop. This condition checking can be carried out at the end of the loop also instead of carrying it out at the beginning. All these situations are possible in C using the **while**, **do-while** and **for** statements.



Apart from these certain other types of program branching are also possible where depending upon the input the program has the capability to branch to many different program segments as in a program menu. Without using many if-else statements such a situation can be handled using the **switch-case-default** construct in C. We will now discuss all these control structures one by one.

### 4.2 The Increment, Decrement and Compound Assignment Operators:

After learning a lot about the relational, logical and conditional operators we now will learn about another set of operators that we will use extensively in our subsequent discussions of loop statements. These are called the increment, decrement and compound assignment operators as given below:

Operator	Description	Examples	Comments
++	Increments the existing value of a variable by 1 ( <i>applicable for integers only</i> )	<code>a++;</code> <code>++a;</code> <code>y=a++;</code> <code>x=++z+6;</code>	<code>x++</code> same as <code>x=x+1</code> <code>++x</code> same as <code>x=x+1</code>
--	Decrements the existing value of a variable by 1 ( <i>applicable for integers only</i> )	<code>a--;</code> <code>--a;</code> <code>y=a--;</code> <code>x=--z+6;</code>	<code>x--</code> same as <code>x=x-1</code> <code>--x</code> same as <code>x=x-1</code>
+=	Increases the existing value of a variable to the left of the operator by an amount to the right	<code>y+=5.2;</code> <code>z+=2+3*y;</code> <code>y+=1;</code>	<code>y+=5.2</code> same as <code>y=y+5.2</code> <code>z+=2+3*y</code> same as <code>z=z+(2+3*y)</code> <code>y+=1</code> same as <code>y=y+1</code> same as <code>y++</code>
-=	Decreases the existing value of a variable to the left of the operator by an amount to the right	<code>y-=4.5;</code> <code>z-=2*b+c;</code> <code>y-=1;</code>	<code>y-=4.5</code> same as <code>y=y-4.5</code> <code>z-=2*b+c</code> same as <code>z=z-(2*b+c)</code> <code>y-=1</code> same as <code>y=y-1</code> same as <code>y--</code>
*=	Multiplies the existing value of a variable to the left of the operator by an amount to the right	<code>y*=k;</code> <code>z*=5+3/y;</code> <code>y*=1;</code>	<code>y*=k</code> same as <code>y=y*k</code> <code>z*=5+3/y</code> same as <code>z=z*(5+3/y)</code> <code>y*=1</code> same as <code>y=y*1</code>
/=	Divides the existing value of a variable to the left of the operator by an amount to the right	<code>y/=k;</code> <code>z/=x+3*y;</code> <code>p/=2;</code>	<code>y/=k</code> same as <code>y=y/k</code> <code>z/=x+3*y</code> same as <code>z=z/(x+3*y)</code> <code>p/=2</code> same as <code>p=p/2</code>
%=	Calculates the mod of an existing value of a variable to the left of the operator by an amount to the right	<code>a%=5*t;</code> <code>z%=5*a+4;</code> <code>y%=7;</code>	<code>a%=5*t</code> same as <code>a=a%(5*t)</code> <code>z%=5*a+4</code> same as <code>z=z%(5*a+4)</code> <code>y%=1</code> same as <code>y=y%7</code>

### Points to note while using the above operators:

The increment operator when used separately does the same job as incrementing the variable to which it is attached by 1. It is usually used in loop applications to increment a counter variable by 1, each time a certain portion of the loop is executed.

**Ex1.:** The follow program section calculates the average of numbers entered through the keyboard:

```
main()
{ int num, count=0, sum=0; float average;
  start loop
  /*Body of Loop*/
  {
    printf("\nEnter integer no.%d to average",count+1);
    scanf("%d", &num);
    sum+=num;
    ++count;
  }
  repeat loop till condition is satisfied

  average=sum/count;
  printf("\nAverage of %d numbers is %f", count, average);
  return(0);
}
```

The variables **sum** & **count** have been initialised to 0

The variable **num** is added to **sum** each time the loop is executed

The variable **count** increments by 1 each time the loop is executed

The variables **sum** and **count** have been initialised to 0 during declaration. Thus when the first number is entered then it is added to the value of **sum** i.e. 0. During subsequent repetition of the loop new numbers will get added to the existing **sum**. The variable **count**, counts the number of terms entered. Finally when the loop terminates, the final **sum** is divided by **count** to get the **average**. Observe how the increment and compound assignment operator have been used to find the sum and increase count. They are same as:

**sum=sum+num and count=count+1**

The **increment operator changes the value of the variable itself by 1**. Thus in the following statements:

```
main()
{ int count=0, y;
  printf("\nThe first value of count is %d", count);
  y=count+1;
  printf("\nThe second value of count is %d", y);
  printf("\nThe third value of count is %d", count);
  ++count;
  printf("\nThe fourth value of count is %d", count);
  y=count+1;
  printf("\nThe fifth value of count is %d", y);
  printf("\nThe sixth value of count is %d", count);
  return(0);
}
```

→ Output = count = 0  
→ y = count + 1 = 0+1=1  
→ Output = y = 1  
→ Output = count = 0  
→ count = count + 1 = 0+1=1  
→ Output = count = 1  
→ y = count + 1 = 1+1=2  
→ Output = y = 2  
→ Output = count = 1

output same

count remains same

It can be seen that though the output of **y=count+1** and **++count** are same for a particular initial value of count, but while **++count** increments the value of **count** by 1 and hence changes the value of the variable itself, the statement **y=count+1** keeps the value of variable **count** same but assigns the value of **count+1** to a second variable **y**. Thus:

**printf("New count=%d",count+1);** → Displays count +1, **no change** in value of **count**  
**printf("New count=%d",++count);** → **Changes** value of **count** by 1 and then displays the result

The above difference should be carefully noted as this may give rise to logical errors in case one is careless in using the increment/decrement operators.

In the above examples the increment/decrement operators were used as expressions, but when used in a statement as *part of a calculation or condition*, then they behave separately depending on where the operator is placed i.e. before or after the variable. That is there is a difference between **++a** and **a++** as shown below:

Case 1	Case 2
<pre>int num=0, a=0; printf("a=%d",a);      → a=0 printf("Num=%d",num);   → Num=0 num=++a; printf("a=%d",a);      → a=1 printf("Num=%d",num);   → Num=1</pre>	<pre>int num=0, a=0; printf("a=%d",a);      → a=0 printf("Num=%d",num);   → Num=0 num=a++; printf("a=%d",a);      → a=1 printf("Num=%d",num);   → Num=0</pre>

In **case 1**, we have used the increment operator before the variable **a**. Thus the function of the statement **num=++a** is the same as:

```
a=a+1; }
num=a; } num=++a → The value of num is the new value of a
```

The variable **a** is first incremented and then it is assigned to the variable **num**.

In **case 2**, we have used the increment operator after the variable **a**. Thus the function of the statement **num=a++** is the same as:

```
num=a; }
a=a+1; } num=a++ → The value of num is the old value of a
```

The variable **a** is first assigned to the variable **num** and then it is incremented.

**Ex2.:** The following program section uses the logical and increment/decrement operators to evaluate an expression as true or false. Note the way in which the expression is evaluated:

```
main()
{ int a=0, b=1, c=2, d;
  c=++a&&++c|++b;
  d=a&&b&&c;
  printf("\nValue of a=%d", a); → Output: Value of a=1
  printf("\nValue of b=%d", b); → Output: Value of b=1
  printf("\nValue of c=%d", c); → Output: Value of c=1
  printf("\nValue of d=%d", d); → Output: Value of d=1
  return(0);
}
```

If the output of the above portion of program segment surprises you, here is the reason why it is such. As mentioned earlier, the value of the variable **c** is evaluated from left to right. First **a** is incremented to 1 by **++a**. The next operator is AND. Hence to test the logic; **c** is then incremented to 3 by **++c**. The result of **logical** ANDing of **++a&&++c** thus gives **1&&3=True=1**. The next operator is OR. But the first parameter of the OR operator has already been calculated to be 1 and hence is sufficient to declare the final output **c** as logically true [since, (1) OR (any-value) = True = 1]. Thus further calculation of the remaining portion of the expression is not carried out and hence **b** is not incremented to 2 by **++b** and remains as 1. The final value of **c** thus becomes True or 1. Hence finally **a=b=c=1**. Next when **d** is calculated, the whole expression is evaluated, as all the operators are logical ANDs, and hence each of the variables is tested to be 1 (which is true in this case) and assigns the logical value of 1 to **d**.

**Note:** There are **no operators as** **+++** or **---** in C. Hence **+++x**, **x+++**, or **y---**, **---y** does not have any meaning *by itself*, though, **z = x+++y** does carry a meaning and is the same as **z=(x++) + y**.

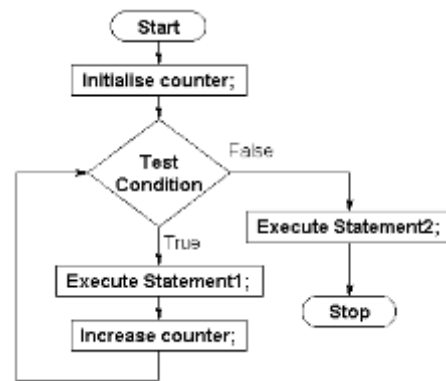
**Ex3.:** The increment operator can give rise to some tricky problems as shown in the following example:

```
main()
{ int a=1, b, c;
  b=a++ + ++a;
  c=a+1;
  printf("\nValue of b=%d", b); → Output: Value of b=4
  printf("\nValue of c=%d", c); → Output: Value of c=4
  return(0); }
```

The expected output for **b** is **b=1+2=3**. But the actual output will be **b=2+2=4**. This is because **++a** will be calculated first and will make **a=2**, then this will be added to **a** of **a++**. *After* the addition, **a++** will make **a=3**. Then *this* **a** will be added to 1 and make **c=3+1=4**.

### 4.3 The while statement:

One of the main techniques used to repeat a part of the program code is by stating a condition to repeat the loop a certain number of times. For doing this C has a set of statements of which the first one is the **while** statement. The flowchart to the right shows the logic of the **while** loop. **Initialisation and increment of the counter are optional** and can be dropped if not necessary. In case the same is used, then do not forget to *initialise the counter before the loop* and *increment the counter inside the loop*. While executing the loop, each time the condition is tested first and if found true, the body of the loop is executed. Else the control goes out of the loop and executes the statement next to the loop.



The structure of the **while** statement is shown below:

```
initialise loop counter;
while (condition)
{
    statements;
    ... ..
    increment loop counter;
}
```

Annotations:

- ← **No semicolon (;) after condition** (points to the condition in the while statement)
- ← **Body of the loop** (points to the statements inside the curly braces)

**Ex4.:** Let us take a simple example to illustrate the functioning of the **while** loop. The following program prints the numbers from 1 to 10.

```
main()
{
    int count=1;
    while(count<=10)
    {
        printf("\nNumber %d", count);
        ++count;
    }
    return(0);
}
```

Annotations:

- ← **Initialisation of the loop counter outside loop** (points to `int count=1;`)
- ← **The condition part checks if the expression within the brackets is true or false** (points to `while(count<=10)`)
- ← **Incrementing the loop counter inside the loop** (points to `++count;`)

In the above program, the number of times the loop should repeat is checked by the variable **count** (you can give any other suitable name to the variable), which has been initialised to 1 during the declaration part. After encountering the **while** keyword, the condition following the keyword is checked. In this case it is checked if `count<=10`. If the condition is satisfied or evaluates to True then the body of the **while** loop is executed. If not, the loop is skipped and the statement immediately *after the body of the loop* is executed.

For the first run of the loop the condition is satisfied since `count=1<10`. The program control enters the body of the loop and prints the value of **count** (you can do any other job as per your requirement). Next it increments the value of **count** by 1 in `++count` and count becomes equal to 2. On reaching the end of the loop block, the program again checks if the condition is satisfied. Since `count=2<10` now, the condition is again satisfied and the block of statements inside the loop again gets executed. This continues till after `++count` is executed **count** becomes equal to 10. This is the last time the loop will be executed, for after this **count** becomes equal to 11 which fails the condition and the loop terminates.

**Ex5.:** We now give below two variations of the above program where we have used the **increment, inside the condition**.

```
main()
{
    int count=0;
    while (++count<=10)
    {
        printf("\nNumber %d", count);
        getchar();
    }
    return(0);
}
```

Annotation:

- ← **Increment used inside the loop condition** (points to `while (++count<=10)`)

**Count** has been initialised to 0. On entering the **while** loop condition it is first incremented to 1 by `++count` and then the condition is tested. The condition will remain true till `++count` makes `count=10` when the loop will be executed for the last time and will print, "Number 10".

In the previous example we have used the operator as `++count`. If we were to use the operator as `count++` then the program will have to be changed in the following way:

```

main()
{ int count=0;
  while (count++<10) printf("\nNumber %d", count);
  getchar();
  return(0);
}

```

At first glance it may seem that the variable **count** has to be initialised to 1 instead of 0, to display the numbers from 1 to 10. But note that when the condition will be tested for the first time, count will be equal to 0 which will satisfy the condition **count<10**. After testing the condition count will be incremented to 1 by **count++**. Thus the value of count that will be printed for the first go will be 1 and not 0. Note that the equal-to sign has been removed and only **<** is used inside the condition. This is because to print 10 as the last number, count will have to be equal to 9 during testing of the condition, so that the **++** after the testing makes it equal to 10. If **<=** were used then in the next go the condition would have been satisfied and the output would have been 11 after the **++** operation on count.

**Ex6.:** We can use the while loop to sum the terms of a G.P. series upto a certain number of terms.

```

main()
{ int sum=0, count=1, current_term=1, terms, a, b;
  puts("\nProgram to find the sum of a G.P. series");
  printf("\nEnter first term of G.P.: "); scanf("%d", &a);
  printf("\nEnter common ratio of G.P.: "); scanf("%d", &b);
  printf("\nEnter number of terms to sum: "); scanf("%d", &terms);
  current_term=a;
  while(count<=terms)
  { sum+=current_term;
    current_term*=b;
    ++count;
  }
  printf("\nThe required sum upto %d terms is %d", (count-1), sum);
  return(0);
}

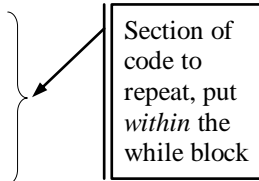
```

**Ex7.:** The while loop is more suited for execution of a loop unknown number of times, depending on certain calculations or user input. One such example is to repeat a certain portion of code depending on whether the *user wants to repeat it*.

```

main()
{ char repeat='Y'; float num;
  while(repeat=='y' || repeat=='Y')
  { printf("\nEnter any decimal number to square: ");
    scanf("%f", &num);
    printf("\nThe square of %f is %f", num, num*num);
    puts("\nPress 'Y' to repeat with a new number");
    repeat=getchar();
  }
  return(0);
}

```



The variable **repeat** is first initialised to 'Y' to enter the loop for the first time. Subsequent runs of the program depend on the input of the user to the prompt to repeat. As long as the user types 'y' or 'Y', the **while** condition gets satisfied and repeats the section of the code within the curly brackets. (Note that the logical OR operator has been used inside the while condition to test if the user has entered 'Y' or 'y', for both of which the condition should be true).

#### **Points to note while using the while statement:**

**Ex8.:** Observe the following section of code to print the numbers from 0 to 32767 using a while construct.

```

main()
{ int i=1;
  while(i<=32767)
  { printf("%d\n", i);
    ++i;}
  return(0);
}

```

The above program will execute indefinitely without ever stopping and will go on printing the numbers. The reason lies in the declaration of the variable `i` as an `int`. After the `printf()` statement prints 32767, `i` should be incremented from 32767 to 32768 by `++i` and the next condition will evaluate to be false as  $32768 \nless 32767$  and the loop should finally stop. But in reality  $32767+1 = -31768$  which is less than 32767 and evaluates to a truth resulting in running of the loop infinitely. To overcome this problem either `i` is to be declared as a long or the program is to be modified in the following way:

```
{ int i=0;
  while(i<32767)
  { printf("%d\n", ++i);
    return(0);
}
```

i initialised to 0 instead of 1

Increment done during printing the output

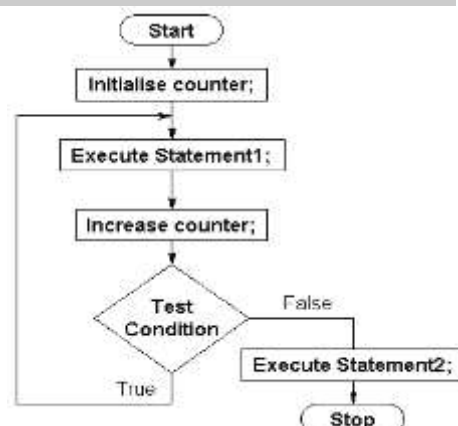
**Ex9.:** Now consider the following program section.

```
{ int i=1;
  while(i<=100);
  { printf("%d\n", i);
    ++i;
  }
}
```

The above loop will run infinitely without showing any output, in other words it will simply hang when run. The reason lies in the placing of a semicolon (;) at the end of the while condition. This is a common programming mistake. This implies there are no segments to execute after the **while**, if the condition is true and the control will come back to test the condition again. But since the variable `i` has not been incremented, the condition will remain true always, `i` being equal to the initial value of 1. Another reason why a loop can run infinitely is if the condition is a **truth always** like `while(1)`.

#### 4.4 The **do-while** statement:

The **do-while** loop is similar to the **while** loop in that it continues as long as the condition remains true. But the main difference is that, while the **while** loop may or may not execute at all depending upon the condition, the **do-while** loop will execute at least once before testing the condition. The reason is apparent from the flow chart depicting the structure of the **do-while** loop.



The condition in a do-while loop is tested at the end of the do-while block and hence the block will execute at least once before testing the condition. The syntax of the do-while loop is:

```
do
{ statements;
  ... ..
} while (condition);
```

No semicolon (;) after do

Body of the loop

Semicolon (;) after while

**Ex10.:** The following program segment tests for a valid input using **do-while** construct.

```
do { puts("\nEnter value of temperature in the Kelvin scale");
    scanf("%f",&temp);
  } while (temp<0);
```

As long as the user enters a negative value, which is an invalid value for a temperature in the Kelvin scale, the while condition is satisfied and the prompt will again ask for an input from the user. When the user inputs a positive value, the while condition will no longer be satisfied and the program will proceed to the next statement after the condition checking. To use a while construct in this case, we would have to initialise the variable `temp` to a positive value to make the while block execute initially.

The above **do-while** construct can also be used to check a valid program input at the time of the input, to avoid any later complications while running the programs.

**Ex11.:** The next program segment is a modified version of the previous program where a different program prompt is displayed based on the tests for a valid input using **do-while** construct.

```
main()
{ char flag='n'; float temp;
  do { if(flag=='n')
        puts("\nEnter value of temperature in the Kelvin scale");
      else
        puts("\nPlease input a valid temperature!");
      scanf("%f", &temp);
      flag='y';
    } while (temp<0);
  return(0);
}
```

A **char** variable called **flag** has been introduced and initialised to 'n', to check if the entered input is valid or not. When the loop runs for the first time, the **if** condition is satisfied and the first **puts()** function is executed. Next the **scanf()** function is executed and finally **flag** is changed to 'y'. If the user enters a valid input then the **do-while** loop is no more repeated. In case the user inputs a negative number, then the **while** condition is satisfied and the loop is again repeated. But this time **flag** has been changed to 'y' and the **if** condition is not satisfied and the **else** executes the second **puts()** function. Further, to limit the number of times the loop is executed for an invalid input, a counter can be used.

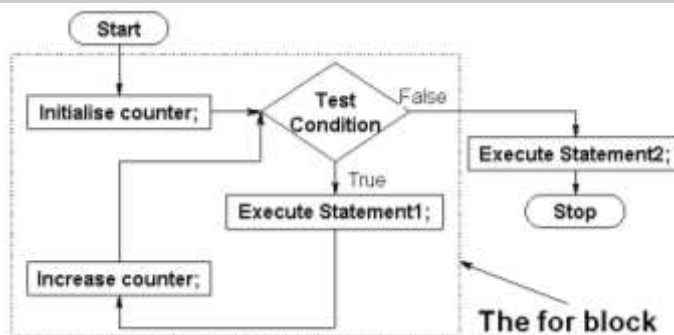
**Ex12.:** Another area where the **do-while** construct can be used is when a certain calculation is to be carried out an unknown number of times based on certain accuracy requirements. The following program calculates the sum of the series  $1+(1/2^2)+(1/3^2)+(1/4^2)+\dots$  upto a certain level of accuracy input by the user.

```
main()
{ int count=1; float sum=0.0, last_sum=0.0, error=0.0, percent;
  printf("\nEnter the percent accuracy required"); scanf("%f", &percent);
  do { sum+=1.0/(count*count);
      error=(sum-last_sum)/sum;
      last_sum=sum;
      ++count;
    } while (error>=(percent/100.));
  printf("\nThe sum of the series upto %d terms is %f", (count-1), sum);
  return(0);
}
```

At each run of the above loop one term is added to the existing sum and the fractional error calculated by calculating the difference between the present sum and last sum and dividing the result by the present sum. The **while** condition then checks if the error is greater than the required one. If so, then again the loop is repeated, an extra term added and the error again checked. This process continues as long as the error is more than the required one. Once the calculated error is smaller than the required one then the while condition becomes false and the loop is terminated.

#### 4.5 The for statement:

The most important of the loop structures is the **for** loop statement. It is the most widely used and can be used in place of the **while** or **do-while** statements with proper modification. The flowchart to the right shows the functioning of the **for** statement. The counter initialisation, condition testing and counter reinitialisation – all can be done within the condition portion of the **for** loop.



**Semicolons (;) separate each section**

```
for(initialise loop counter; check condition; reinitialise counter)
{ statements;
  ... ..
}
```

**Body of the loop**

**No semicolon (;)**

The **for** statement executes in the following stepwise manner. Counter is set to initial value. Condition is tested. If condition is found true then body of loop is executed. Counter is incremented by increment value. Condition is tested and if found true then body of loop is executed. The above process is repeated till the condition is found false.

**Ex13.:** The next program uses the **for** loop to print the Fibonacci series 1, 1, 2, 3, 5, 8, ... to 10 terms.

```
main()
{ int i, last_term=1, sec_last_term=0, current_term;
  printf("\nProgram to print the Fibonacci series upto 10 terms");
  printf("\nTerm 1 is = 1");          → Prints the first term of the series

  for(i=2;i<=10;++i)                → Prints the subsequent terms starting from Term 2
  { current_term = sec_last_term + last_term;
    printf("\nTerm %d is = %d", i, current_term);
    sec_last_term=last_term;
    last_term=current_term;
  }
  return(0);
}
```

The first two terms of the Fibonacci series being the same, the first term is printed outside the loop. The next 9 numbers are printed using the **for** loop to generate each successive term using the algorithm:

**Current term = (second last term) + (the last term)**

Since the first term has already being printed, the loop counter **i** has been initialised to 2 and is used to count the terms printed. After printing each new term, the counter variable **i** is increased by **++i**. This process continues till **i** becomes equal to 10, when the last i.e. 10<sup>th</sup> term is printed. Then **i** increases to 11, which makes the test condition **i<=10** false and the loop terminates.

**Ex14.:** The following program uses the **for** loop to print a conversion table from Centigrade to Fahrenheit, the starting and ending temperatures being input by the user.

```
main()
{ int i, centigrade, cen_start, cen_stop; float fahrenheit;
  printf("\nEnter the temp. in Centigrade to start"); scanf("%d", &cen_start);
  printf("\nEnter the temp. in Centigrade to stop"); scanf("%d", &cen_stop);

  for(i=cen_start; i<=cen_stop; ++i)
  { Fahrenheit=(9.0/5.0)*i+32;
    printf("\n%d deg. C is = %.2f deg. F", i, Fahrenheit);
  }
  return(0);
}
```

In the above example both the counter initialisation value and the condition checking value have been input by the user and are variable terms instead of literals.

**Ex15.:** In the above two examples we have incremented the loop counter at each run of the loop by a value of 1. In general the counter can be incremented by any value as shown in the example below to calculate the A.P. series 1, 5, 9, 13, 17 ...

Before going for a solution note that the  $n^{\text{th}}$  term of the series is given by  $[4*(n-1)+1]$  where 4 is the common difference. Hence the value of the counter **n** should go upto a value of  $[4*(n-1)+1]$  and since the common difference is 4, during increment of the counter, we are incrementing it by 4 using the expression **n+=4**. Though there are better ways to carry out the generation of the above series, but just to illustrate the fact that the increment of the counter in the **for** loop can be any value we have given the following example.

```
main()
{ int n, count=1, a=1, b=4, term;
  printf("\nProgram to print the A.P. series upto n terms");
  printf("\nInput the number of terms to find"); scanf("%d", &term);

  for(n=a; n<=(a+(term-1)*b); n+=b)
    printf("\nTerm %d is = %d", count++, n);
  return(0);
}
```



**Ex16.:** The next example shows the **nesting of for loops** by calculating the value of the exponential series 'e', given by the relation:  $e = 1 + 1/(1!) + 1/(2!) + 1/(3!) + 1/(4!) + \dots$  to n terms.

```
main()
{ int n, i, j;
  double sum = 1.;
  double factorial = 1.;
  puts("\n*** PROGRAM TO CALCULATE e = 1+ 1/1! +1/2! +1/3! +... ***");
  printf("\nInput number of terms to include: "); scanf("%d", &n);

  for(i=1; i<=(n-1); i++)
  { factorial=1.;
    for(j=1; j<=i; j++)
      factorial*=j;
    sum+=1.0/factorial;
  }
  printf("\nThe sum for %2d terms is \t%.9lf", n, sum);
  return(0);
}
```

This for counts the number of the term given by i, to continue upto n.

This for calculates the actual factorial value of the i<sup>th</sup> term

Let us now examine the functioning of the above program section by taking the number of terms to calculate the sum as 4. The first term i.e. 1 is being taken care of in the initial value of **sum**. The computation of the next terms are being carried by the subsequent **for** loops.

i	j	factorial	sum
1	1	1	1 + 1! = 2
2	1	2!	2 + 1/2! = 2.5
	2		
3	1	3!	2.5 + 1/3! = 2.666666667
	2		
	3		

Since the first term has been taken care of in the **sum**, the number of terms counted by **n** goes up to **(n-1)** in the first **for** loop. Then for each value of **i**, **j** is used to calculate the value of the factorial of the current value of **i**. After the factorial is calculated by the second **for** loop, the term  $1/i!$  is added to the existing **sum** to generate the total sum. This process continues until the last term determined by **(n-1)** is reached.

**Ex17.:** Another common example of using nested loops is to print the multiplication tables from 2 to 10.

```
main()
{ int row, column, product;
  puts("\n*** MULTIPLICATION TABLE ***");

  for(row=1; row<=10; row++)
  { for(column=1; column<=10; column++)
    { product=row*column;
      printf"%4d", product);
    }
    printf("\n");
  }
  return(0); }
```

For each value of row, column will vary from 1 to 10 and print the product of row\*column. Say for row=3, the column will vary from 1 to 10 and hence will print the product from  $3*1=3$  to  $3*10=30$  in a row. In this way the whole multiplication table will be printed on the screen.

### **Multiple initialisation in the for loop: The use of the Comma (,) operator:**

The comma operator in general allows two expressions to appear at a place, where otherwise only one expression would have been used. The general syntax of the use of comma operator is shown below:

**for (exp1a, exp1b, ...; exp2; exp3a, exp3b, ...)**

Multiple **initialisation**  
separated by commas

Single test of  
condition

Multiple **reinitialisation**  
expressions separated by commas

The expressions **exp1a**, **exp1b**, etc. separated by commas are used to initialise different variables simultaneously within the loop. Similarly expressions **exp3a**, **exp3b**, etc. separated by commas are used simultaneously within the loop to **reinitialise the variables**. Though multiple statements can be used in the initialisation and in the reinitialisation portions, but **only one test expression** is allowed, as given by **exp2**. The single test expression can however be an expression containing several variables connected by logical operators and evaluating to a particular value.

**Ex18.:** The following program shows one use of the comma operator to calculate the sum of the series:  
 $1 - 3 + 5 - 7 + 9 - 11 + \dots$  up to  $n$  terms.

```
main()
{ int i, term, sign, sum=0;
  puts("##Program to find the sum of the series 1-3+5-7+9-11+...##");
  printf("\nEnter the number of terms to sum for the series: ");
  scanf("%d", &term);
  for((i=1, sign=1); i<=term; (i++, sign*=-1))
  { sum+=(1+(i-1)*2)*sign; /*n-th term is a+(n-1)b*/
    printf("\nTerm %d = %d", i, (1+(i-1)*2)*sign);
  }
  printf("\n\nThe sum of the series upto %d terms is %d", term, sum);
  return(0);
}
```

Comma (,) used to separate the two variables

The above program functions in the following manner. If the signs of the different terms of the series are ignored then the series can be seen to be a simple A.P. series with first term equal to 1 and the common difference equal to 2. Hence the  $n^{\text{th}}$  term of the series will be  $a+(n-1)b=1+(n-1)*2$ . Next comes the sign of the terms. It can be observed that each term has a sign  $(-1)$  times the sign of the last term. Thus since the first term has a positive sign, the signs of the subsequent terms can be calculated by multiplying each new term by  $(-1)$  times the sign of the last term. If the last term had a positive sign the new term will have a negative sign and vice versa.

Inside the body of the **for** loop, this logic is been utilised to generate the successive terms. The first counter variable **i** is being initialised to 1 and used to count the number of terms and the variable called **sign** has been initialised to 1 also and is used to determine the sign of the current term. After each run of the loop, **i** is increased by 1 by **i++** and the **sign** is being simultaneously multiplied by  $(-1)$  to generate the sign of the next term and is being multiplied with the new term to get the new term with proper sign. Adding the new term with proper sign to the last sum generates the sum.

#### 4.6 The **break** statement:

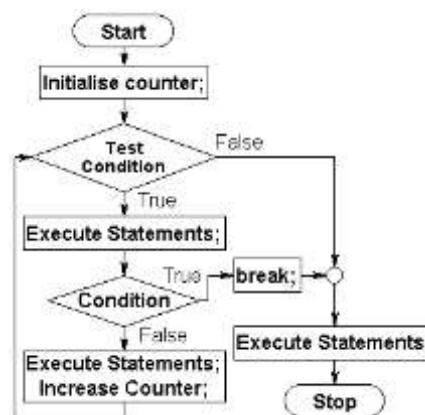
The **break** statement is used to forcibly come out of a program loop bypassing the normal loop condition test or to terminate a **case** in the **switch** statement (to be discussed later). The program control comes out of the loop and the statement immediately next to the loop gets executed. The flowchart to the right explains the working of the **break** statement. The condition to break out of a loop is generally checked by using an **if** statement. In case of nested loops, the control comes out of the loop in which the break statement is located and not from all the outer loops.

The following example illustrates the use of the **break** statement. Numbers are entered using an infinite **while** loop. To finish entry, the user types a negative number to activate the **break** condition.

```
{ int count=0; float sum=0.0, num;
  printf("\nEnter numbers to average. Enter a negative number to end");
  while(1)
  { scanf("%f", &num);
    if(num<0.0)
      break;
    sum+=num;
    count++;
  }
  printf("\nAverage of %d numbers is %f", count, sum/count);
  return(0);
}
```

→ Note: the condition of the **while** statement is made always true to continue the loop as long as the user desires.

→ The **break** statement helps to exit the infinite **while** loop when the user enters a negative number



Instead of using the infinite **while** loop, the user can also use the infinite **for** loop with the condition of the **for** loop set as shown, without any counter or condition to check:

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

#### 4.7 The *continue* statement:

The *continue* statement is used to take the control to the beginning of the loop, bypassing certain statements within the body of the loop, based on some conditions usually given by an *if* statement. The following example prints the capital and small letters of the alphabet using a single loop and the *continue* statement:

```
main()
{ int i;
  puts("## Program to print the Capital and Small alphabets ##\n");
  for((i=65; i<=122; i++)
    { if( i>=91 && i<=96 )
        continue;          → The continue statement helps skip the values from 91 to 96
        printf("%c, ", i);    which do not represent any alphabet
    }
  return(0);
}
```

The next example prints the terms of the series 1, 3, 4, 7, ... Starting from the third term, each term is the sum of the last two terms of the series. Thus the current term is computed by adding the last two terms. However for printing the first two terms, this procedure is not required as the first two terms are given. Thus the program skips the computation part for the first two terms by using the *if* condition and the *continue* statement. The program runs even if someone enters *n*=1, i.e. wants to print only the first term.

```
#include <stdio.h>

int main()
{ int n, i, second_last_term=1, last_term=3, current_term;
  printf("\nProgram to print n terms of the series 1, 3, 4, 7, ...");
  printf("\nInput the number of terms of the series to print: ");
  scanf("%d",&n);

  for(i=1; i<=n; i++)
  { if(i==1)
      {printf("\n%d, ", second_last_term);
        continue;          → The continue helps skip the rest of the block
      }
    if(i==2)
      {printf("%d, ", last_term);
        continue;          → The continue helps skip the rest of the block
      }

    current_term = last_term + second_last_term;
    second_last_term = last_term;
    last_term = current_term;
    printf("%d, ", current_term);
  }

  return 0;
}
```

#### Output of the above program when run:

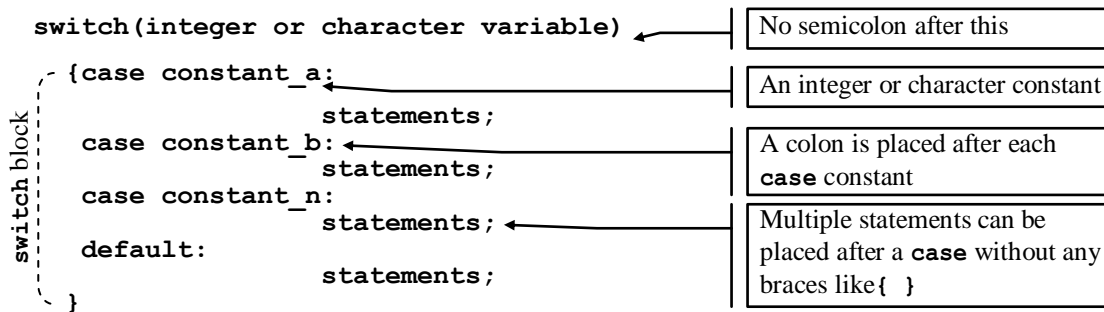
```
Program to print n terms of the series 1, 3, 4, 7, ...
Input the number of terms of the series to print: 5
1, 3, 4, 7, 11,
```

**Note:** *Break* and *Continue* when used properly, work faster than the corresponding structure techniques.

#### 4.8 The *switch-case-default* statement:

To select from multiple branching, apart from using nested *if-else* statements, C provides us with another special type of construct called the *switch-case-default* construct. The particular construct successively tests the value of an expression or variable against a list of integers or character constants. When a match is found, the statements associated with that constant are executed.

However *switch-case* can only check for an equality and a *case* can have only integer or character constants (floats are not allowed), whereas *if-else* can check for any logical or relational expression and can include a float value. The general syntax for the *switch-case* construct is given in the next page:



When a **switch** statement is run, the value contained in the integer or character variable after the **switch** keyword is matched with each of the constant values following the **case** keywords. As soon as the **value matches with a case constant, the statements following that case, and all subsequent cases along with the default statement get executed**. If no match is found, then only the statements following the **default** keyword (which is optional) get executed.

To execute only the statements following a particular **case**, a **break** statement is included after the statements following the **case** constant. Once a **case** is satisfied, the control simply falls through the subsequent **cases** till it reaches a **break** statement. The following program illustrates the use of the **switch-case-default** construct. The program calculates the total number of tea, coffee or cold drink items sold from a sale counter by prompting the user for each sell.

```

main()
{
    char prompt='x';           → prompt is assigned a value 'x' to start the while loop
    int tea=0, coffee=0, drink=0;
    printf("\nEnter T for Tea, C for Coffee, D for ColdDrink and E to Exit:\n");
    while( (prompt!='e') && (prompt!='E') ) → The while loop continues till a user types e or E
    {
        prompt=getchar();
        switch(prompt)          → The value in the variable prompt will be checked
        {
            case 't' :
            case 'T' : ++tea;    → Control halts here if prompt = 't' or 'T'
                        break;   → After incrementing tea, control breaks out of switch
            case 'c' :
            case 'C' : ++coffee; → Control halts here if prompt = 'c' or 'C'
                        break;   → After incrementing coffee, control breaks out of switch
            case 'd' :
            case 'D' : ++drink;  → Control halts here if prompt = 'd' or 'D'
                        break;   → After incrementing drink, control breaks out of switch
            case '\n':
            case 'e' :
            case 'E' : break;    → In case of 'e', 'E' or '\n', control breaks out of switch
            default : printf("\nEnter a valid choice!"); → In case of an invalid entry, the
                                                                default statement is executed
        }
    }
    printf("\nTotal Sale: Tea=%d, Coffee=%d, ColdDrink=%d", tea, coffee, drink);
    return(0);
}

```

Outer while block is indicated by a dashed line on the left.

Inner switch block is indicated by a dashed line on the left.

In the above program, the user inputs the type of drink, i.e. tea, coffee or cold drink, into the variable **prompt**. The variable **prompt** is then used for checking the option entered by the user, by placing it as the condition variable for the **switch** statement.

If the user inputs say 'C', the value in **prompt** is 'C'. This value is then checked with each of the **case** constant values i.e. 't', 'T', 'c', and finally when the **case** constant 'C' is encountered, the condition gets satisfied and the statements following 'C' get executed i.e. the value of **coffee** is incremented by one and the control **breaks** out of the **switch** block.

After this the **while** condition is again checked and the process repeated till the user enters 'e' or 'E', when the **while** condition gets false and the program prints the total sale and quits.