

Chapter -2

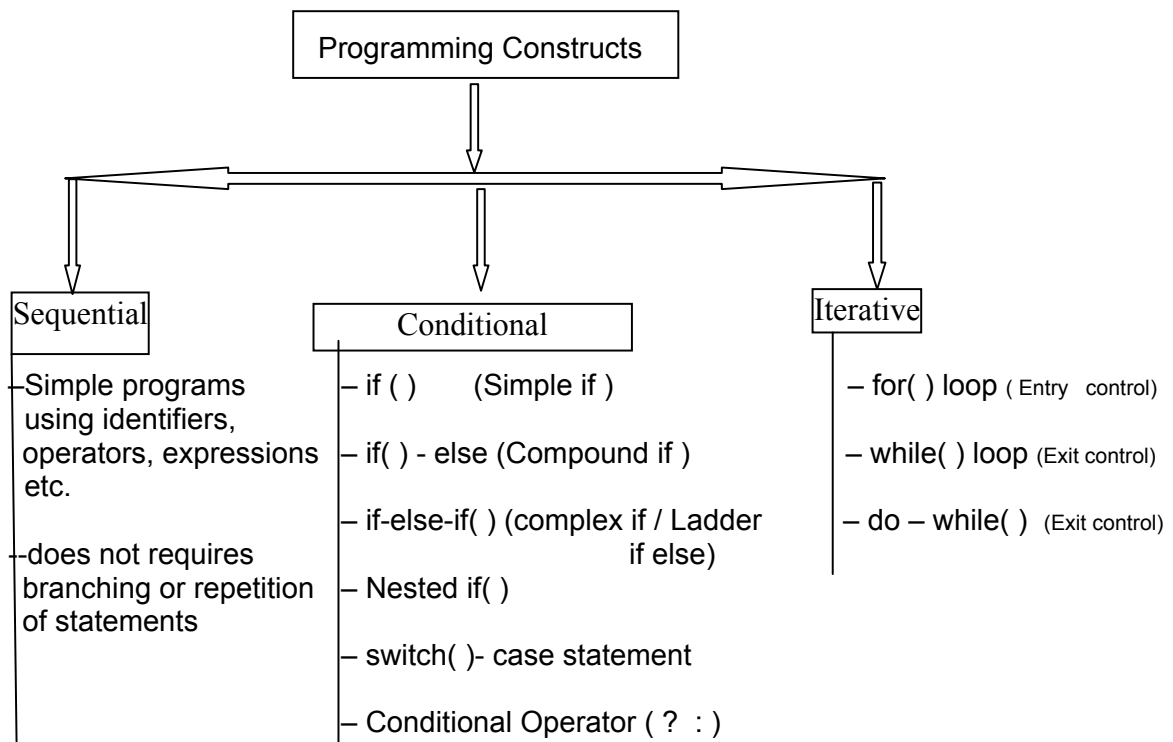
Using C++ constructs

Objectives:

- to analyze syntaxes of various programming constructs available in C++.
- to draw comparison between various programming constructs.
- to apply the syntax of various programming constructs in problem solving

2.1 Categories of available constructs

After exploring into various types of flow of control / logic in different programming situations let us go through the detailed syntax/format of each of the programming constructs available in C++, using which we can monitor flow of control in our program. Here is one diagram which categorizes C++ constructs in detail :



Let us deal each one of them one after another in detail.

2.1 Conditional Constructs :

2.1.1 : Simple If ()

syntax :

```
if ( <conditional expression / logical statement> )
{
    // statements to be executed when logical statement is satisfied
    // i.e. when the logical statement yields a true value
}
```

points to remember :

- i) a logical statement always evaluates as true / false.
- ii) any value in C++ other than zero (positive / negative) is considered to be true whereas a zero (0) is considered to be false.
- iii) < > in syntax is known as a place holder, do not type it while writing program. It only signifies that any thing being kept there varies from program to program.
- iv) if there exists only one line of program statement under if() scope then we may omit curly braces { }

The statement kept under simple if () gets executed only when the conditional expression/logical statement under it is evaluated as true.

Examples :

```
int x = 1 , y = 3;
x += y;
if ( x > y )
{
    cout<<"x is greater";
}
```

In the above example the conditional statement under if () will be always evaluated as true because the value of x will become 4 before the comparison thus the expression $4 > 3$ yields a true value letting the statement under if() to execute i.e. the output of the above code snippet would be " x is greater "

Workout yourself :

Convert the above code snippet into a program in your practical period and execute it, verify the ☐ output.

Now change the initial value of x to 0 instead of 1 i.e. Change `x = 0` in the program. ☐ Execute the

2.1.2 : Compound If () : if – else combination

syntax :

```
if ( < conditional statement > )
{
    // statements to be executed when logical statement is satisfied
    // i.e. when the logical statement yields a true value
}
else
{
    // statements to be executed when logical statement is not satisfied
    // i.e. when the logical statement yields a false value
}
```

Example :

```
int x = 0 , y = 3;
x += y;
if ( x > y )
{
    cout<<"x is greater";
}
else
{
    cout<<"we are in else part because x and y both became equal.";
}
```

The above code snippet (portion) has two different paths of execution. If the conditional statement under if() is evaluated to be true then the statement under if () block will be executed otherwise the statements under else block would be executed. The above code produces an output as “we are in else part because x and y both became equal.” because the conditional statement under if () evaluates as false as x is not greater than y, it is same as that of y.

Workout yourself :

In the above code snippet modify the logical statement under if () such that the if () block gets executed instead of else() block. You should not modify the initial values of x and y or change number of variables in the program.

2.1.3 : Complex If () : if – else ladder

syntax :

```
if ( <condition -1> )
{
    // do something if condition-1 is satisfied
}
else if ( <condition – 2 >)
{
```

```

        // do something if condition -2 is satisfied
    }
    else if (<condition – 3 >)
    {
        // do something if condition- 3 is satisfied
    }
        :
        :    // many more n-1 else - if ladder may come
        :
    else if( < condition – n >)
    {
        // do something if condition – n is satisfied
    }
    else
    {
        // at last do here something when none of the above else-if( )
        //conditions gets satisfied.
    }

```

In the above syntax there are ladder of multiple conditions presented by each if() , all of these conditions are mutually exclusive that is only one of them would get satisfied and all the conditions below it would not be evaluated and is discarded.

Say suppose if condition-3 gets satisfy i.e. it yields a true value for the condition, the statement under the block of third if() gets executed and all other n number of if() conditions below it would be discarded.

If none of the n if() conditions gets satisfied then the last else part always gets executed. It is not compulsory to add an else at the last of the ladder.

Example :

```

char ch = getch( );
if ( ch >= 'a' && ch <= 'z' )
{
    cout<<"you have inputted a lowercase alphabet";
}
else if ( ch >= 'A' && ch <= 'Z' )
{
    cout<<"you have inputted an uppercase alphabet";
}

```

```

else if ( ch > '0' && ch <= '9' )
{
    cout<<"you have inputted a digit";
}
else
{
    cout<<"you have inputted any special character or symbol";
}

```

In above code snippet a character is being inputted from user and is being checked upon by various if () condition as alphabets, digit or any other special symbol. If the first condition gets satisfied then the character inputted is a lower case alphabet, if not then the second if () is evaluated , if the second if() gets satisfied then the character is an upper case alphabet, if not then the third if () is being evaluated , if it is satisfied then the character is a digit if not then finally it is inferred as any other symbol in the last else() .

The benefit of this type of conditioning statement is that we can have multiple conditions instead of just having one or two as seen in case of earlier if() constructs.

Workout yourself :

A date in the format of dd/mm/yyyy when dd, mm, and yyyy are inputted separately is said to be a valid date if it is found in a particular year's calendar. For example 22/12/2012, 29/02/2012 are valid dates in calendar of 2012, but 31/06/2012 or 30/02/2012 are invalid dates in calendar of 2012.

Find out how many ladders of if() conditions would be required to write a program for checking validity of a date.

Discuss your answer with your teacher.

2.1.4 : Nested if-else

You might have seen a crow's nest or any other bird's nest , the materials being used to build the nest are enclosed within one another to give ample support to the nest.

We also have the same concept of nesting of constructs within one another , to give ample strength to our program in terms of robustness , flexibility and adaptability (these terms you have learned in Unit - 3 earlier.)

We are now considering nesting of an if () construct within another if () construct i.e one if() is enclosed within the scope of another if(). The construct thus formed is called nested if(). Let me show you few of the syntax forms of nested if() :

Syntaxes: **Simple if () nested within scope of another simple if ()**

```
if ( <outer- condition > )
{
    if ( <inner-condition> )
    {
        //some statements to be executed
        // on satisfaction of inner if ( ) condition.
    } // end of scope of inner if( )
```

```

//some statements to be executed
// on satisfaction of outer if ( ) condition.

} // end of the scope of outer if( )

```

compound if () nested within scope of another compound if ()

```

if ( <outer- condition > )
{
    if ( <inner-condition> )
    {

        //some statements to be executed
        // on satisfaction of inner if ( ) condition.

    }
    else
    {
        // statements on failure of inner if( )
    }

    //some statements to be executed
    // on satisfaction of outer if ( ) condition.

}
else
{

    // statements on failure of outer if( )

}

```

Ladder if-else-if () nested within scope of another ladder if-else-if ()

```

if ( <outer- condition-1 > )
{
    if ( <inner-condition - 1> )
    {

        //some statements to be executed
        // on satisfaction of inner if ( ) condition.

    }
    else if ( <inner-condition – 2> )
    {
        // statements on failure of inner if( )
    }
    else
    {
        // last statement of the inner ladder if-else-if
    }
}

```

```

        //some statements to be executed
        // on satisfaction of outer if ( ) condition.

    }
    else if ( <outer-condition-2 >)
    {

        // statements on failure of outer if( )

    }

```

Like this you may try to keep any type if () construct within the scope of other type of if () construct as desired by the flow of logic of that program.

Workout yourself:

- i) Try to nest simple if () upto 4 levels ii) Try to nest complex if within a compound if ()
- iii) Try to nest three simple if () within another simple if()

let us now explore few example programs using various if() constructs :

program 2.1

// program to compare three integer values to find highest out of them

```

#include<iostream.h>
void main( )
{
    int n1= n2 = n3 = 0;
    cout<<"Input three integers";

    if( n1 > n2 && n1 > n3)
        cout<<n1 << " is the highest value";
    else if( n2 >n1 && n2 > n3)
        cout<< n2 << " is the highest value";
    else if ( n3 > n1 && n3 > n2 )
        cout<< n3 << "is the highest value";
    else
        cout<< "all values are equal";

}

```

Explanation :

Lets parse the logic of the above code (shaded area) to understand it using a dry run :

Case – I

let us assume that the value of **n1 = 1 , n2 = 5 and n3 = -7**

evaluation of the first if() in the ladder :

```

if( n1 > n2 && n1 > n3 )
==> if( 1 > 5 && 1 > -7 )
==> if ( false && true )
==> if( false ) // as per truth table of logical and ( && ) operator

```

since the first condition is evaluated as false so the next condition in the ladder would be evaluated

```

if( n2 > n1 && n2 > n3 )
==> if ( 5 > 1 && 5 > -7 )
==> if ( true && true )
==> if(true) // as per truth table of logical and ( && ) operator

```

since the second logic is evaluated as true, it open the block of second if () and the statements under the second else-if () block gets executed and an output is printed on the console as :
 "5 is highest value"

Workout yourself :

- i) Try to execute the above program 2.1 using dry run method with values n1 = 1 , n2 = 1 and n3 = 1
- ii) Try to implement the above program using any other type of if() construct

program 2.2

// program to find whether a 4 digit inputted year is a leap year

```
#include<iostream.h>
```

```
void main( )
```

```
{
```

```
    int year = 0 ;
```

```
    cout<<"Input a 4 digit year"
```

```
    cin>> year;
```

```
    if ( year % 4 == 0 )
```

```
    {
```

```
        if ( year % 100 == 0 )
```

```
        {
```

```
            if( year % 400 == 0)
```

```
            {
```

```
                cout<< "Year : " << year << " is a leap year";
```

```
            }
```

```
        else
```

```
        {
```

```
            cout<<"Year : " << year << " is not a leap year";
```

```
        }
```

```
    }
```

```
    else
```

```
    {
```

```
        cout<<"Year : " << year << " is a leap year";
```

```
    }
```

```
}
```

```
else
```

```

{
    cout<<"Year : "<< year << " is not a leap year";
}

```

```

} // end of main ( )

```

The above program is a very good example showing the use of nesting of if ().

Explanation:

Let us first understand which year would be called leap year , a leap is a year which :

- i) is divisible by 4 but not divisible by 100
- ii) is divisible by 4 as well as divisible by 100 and at the same time divisible by 400

Any other criteria will make the year as a non-leap year candidate.

Dry Run -1

Let us parse the gray area code of the above program with a dry run having year = 1994

```

if( 1994 % 4 == 0)
==> if ( 2 == 0 )
==> if ( false )

```

Since the first if condition is not satisfied the flow of the program proceeds to its else block and prints the output as " Year : 1994 is not a leap year"

Dry Run – 2 :

Let us parse the gray area code of the above program with a dry run having year = 2000

```

if( 2000 % 4 == 0)
==> if ( 0 == 0 )
==> if ( true )

```

Since the first condition evaluates out to be true it opens up its block and the next statement in the inner block gets executed as :

```

==> if (2000 % 100 == 0)
==> if ( 0 == 0 )
==> if ( true )

```

Since the first nested if () condition evaluates out to be true it opens up its block and the next statement in the inner block is continues as :

```

==> if( 2000 % 400 == 0)
==> if ( 0 == 0 )
==> if ( true )

```

Since the second nested if () condition evaluates out to be true it opens up its block and the next statement in the inner block is continues to print output on console as :

"Year : 2000 is a leap year"

Workout yourself:

- i) Google out to find that whether year 1900 was a leap year or not and then verify it using a dry run with the help of above program.
- ii) Try to implement the above code without using nested if () construct.

Check your progress:

1. Find error in code below and explain.

```
#include<iostream.h>
```

```
void main()
```

```
{
```

```
    int x = 5 ;
```

```
    if( x > 5 )
```

```
    {
```

```
        x += 5;
```

```
        int y = 8;
```

```
    }
```

```
    y += x;
```

```
}
```

2. Find the output of the code below :

```
void main( )
```

```
{
```

```
    int NoOfGirls = 4;
```

```
    int NoOfBoys = 10 ;
```

```
    if ( NoOfBoys = 8  &&  NoOfGirls <= NoOfBoys )
```

```
        cout<<"Great achievement";
```

```
    else
```

```
        cout<<"Greater achievement";
```

```
}
```

3. Find the output of the code below :

```
void main( )
```

```
{
```

```
    int circle = 5 , rectangle = 0 , square = 4 , triangle = 0 ;
```

```
    if( circle)
```

```
    {
```

```

        if( rectangle || square )
        {
            cout<<"Draw diagram";
        }
        else if( ! rectangle && ! square )
        {
            cout<<"Invalid diagram";
        }
        else
        {
            if( circle == rectangle || square == triangle )
            {
                cout<<"Canvas Available";
            }
        }
    }
    cout<<"Invisible diagram";
}

```

4. Find the output of code below:

```

void main( )
{
    int x = 3 , y = 5;
    if( x <= 5 );
        cout<<"Hurray";
    else
        cout<<" Trapped";
}

```

5. Write a program which inputs day(dd) , month(mm) and year(yyyy) and checks whether it is a valid date or not. [A valid date must lie on the calendar]
6. Dipu Jewellers gives discount on a fixed purchase total based on following criteria :

Offer	Offer Months	Discount %	Purchase total
Winter Bonanza	Oct to Feb	30	Between 3000 to 5000
Summer Bonanza	Mar to June	20	Between 10000 to 12000
Monsoon Bonanza	July to Sep	10	Between 2000 to 10000

If the purchase amount of a customer does not lies between the given purchase total then no discount should be given.

Write a program to calculate the Bill amount of a purchase after giving proper discount.

2.1.5: switch-case statement

switch-case construct is a type of conditional construct which resembles the same flow of logic as that of ladder if-else-if with few exceptions. Let us observe the syntax of the switch-case construct of C++ :

Syntax :

```
switch ( <switching variable | switch expression > )
{
    case <value-1> :
        //do something if switching variable value matches
        // with that of value-1
        break;
    case <value-2> :
        //do something if switching variable value matches
        // with that of value-2
        break;
    case < value-3> :

        //do something if switching variable value matches
        // with that of value-3
        break;
        :
        :
        :
    case <value-n> :
        //do something if switching variable value matches
        // with that of value-n
        break;
    default :
        //do something if switching variable value does not matches
        // with any value between value-1 to value-n
}
```

A switch-case control matches a particular switching value in a switch variable or a value generated after evaluating an expression with several values, when it finds an exact match between switch variable value and case value it enters into that particular case to execute the statements under that case, and then on finding a break statement it jumps out of the case and then comes out of the scope of switch-case without considering other cases value.

Example:

```
int menu_choice = 0;
cin>>menu_choice ;
switch (menu_choice)
{
```

```

case 1 :
    cout<< "press key 's' to start the game";
    break;
case 2 :
    cout<<"press key 'n' to navigate through game plan";
    break;
case 3 :
    cout<<"press key 'c' to change the level of the game";
    break;
case 4 :
    cout<<"press key 'f' to fast your speed";
    break;
case 5 :
    cout<<"press key 'x' to exit from game";
    break;
default :
    cout<<"you have to choose between 1 to 5" ;
}

```

The above code implements a game menu design where inputted integer value in the switching variable menu_choice is being matched with values 1 to 5 , each value representing one game action.

You may observe that the flow of the switch-case construct is same as ladder if-else-if construct, because it also deals with multiple paths of execution out of which a selected path gets executed. If none of the conditions gets satisfied then the statements under the default case gets executed.

Points to remember while using a switch-case construct in a program :

- a) a **break statement** always ends a case, if break is not placed then all the cases after the current case gets executed till a break statement is met.
- b) a switch-case always matches its switch variable value with a single constant case value, this case value must always be a single integer value or a single character. Floats and double values of switch variable are not valid. i.e. the code below is invalid code :

```

float v = 0.0;
cin>> v;
switch( v)
{
    case 1.1 :
        // do something
        break;
    ...
}

```

Invalid switch variable declaration

Invalid case value

- c) the cases in the switch cannot provide a range of values to be matched with the

switching variable i.e. we can't write statements like `>= 4 && <= 9` with cases. The following code is invalid :

```
char ch = '*';
cin >> ch;
switch(ch)
{
    case >='a' && <='z'

        cout<<"You have entered lowercase alphabet";
        break;
    case >= 'A' && <= 'Z'
        cout<<"You have entered uppercase alphabet";
        break;
}
```

The above code gives us an example where ladder if-else-if has an advantage or switch-case, because we can have a logical statement having a range of values to be compared.

- d) The sequence of case value does not matters i.e. It is not compulsory to keep a lower case value as first case and the highest case value being the last case. They can exist in any order.
- e) The default case is optional and should be always kept at the last place in switch-case construct
- f) A switch-case construct can also be nested within another switch-case operator.

Workout yourself :

- i) Write a program which inputs a month number from user and then finds the total days present in that month. Use only switch-case construct. For example if user inputs month number as 2 then the program displays "February has 28 or 29 days".
- ii) Compare and contrast switch-case construct with ladder if-else-if construct in a tabular form.

2.1.6 : Conditional operator (< > ? <> : <>)

Conditional Operator is a small short hand operator in C++ which helps to implement flow of logic based on some condition like if-else construct. The syntax of the conditional operator is :

Syntax :

```
<logical expression> ? < true part > : < false part > ;
```

The logical expression in the operator gets evaluated either as true or false , if true then statement after symbol (?) gets executed otherwise statement after symbol (:) gets executed. It acts much like an if () - else construct but can only execute single statement. Like if-else the conditional operator

cannot have a block of code to execute in its true or false part.

We often use conditional operator to implement a short one line conditional expression.

Example:

```
int x = 5 , y = 7;  
int result = ( x > y ) ? 1 : 0;  
cout<< result;
```

The output of the above example would be 0 because the condition $x > y$ is not being satisfied , hence the value 0 is being assigned to result. A conditional operator can be also nested within another conditional operator.

Workout yourself:

Write a program using conditional operator to find the highest of three inputted integers.
[Hint : use nesting of conditional operator]

2.1.7 : Nesting of all conditional constructs

In real life programming situations often all the conditional constructs are being used in a single program. Any of the conditional construct can be nested within any of the other construct. For example a switch-case construct can nest a ladder if-else-if within its scope, or each ladder condition of a ladder if-else-if may nest a switch-case construct within its scope. The following code snippet justifies this idea :

```
switch( < an expression> )  
{  
    case <value-1> :  
        if( < condition -1 > )  
        {  
            // some code  
        }  
        else if (< condition-2> )  
        {  
            // some code  
        }  
        else  
        {  
            // some code  
        }  
        break;  
    case <value-2 >  
        if( < condition -1 > )  
        {  
            // some code  
        }  
        else if (< condition-2> )  
        {  
            // some code  
        }  
}
```

```

        else
        {
            // some code
        }
        break;
        ...
    } // end of switch-case

```

Similarly we can have :

```

if ( <condition -1 >)
{
    switch(var1)
    {
        case <value1> :
            ...
            break;
        case <value2>

```

```

            ...
            break;
        case<value-3>
            ...
            break;
    }
    else if ( <condition-2>)
    {
        switch(var1)
        {
            case <value1> :
                ...
                break;
            case <value2>
                ...
                break;
            case<value-3>
                ...
                break;
        }
    }
    ...

```

Ask your teacher which of the constructs can be nested within which of the other constructs.

Check Your Progress :

1. Find the output of the code given below :

```
#include<iostream.h>
void main( )
{
    int x = 3;
    switch(x)
    {
        case 1 :
            cout<< "One";
            break;
        case 2 :
            cout<< "Two";
            break;
        case 3 :
            cout<< "Three"
        case 4 :
            cout<< "Four";
        case 5:
            cout<< "Five";
            break;
    }
}
```

2. Find error in following code :

```
#include<isotream.h>
void main()
{
    int a = 10;
    int b = 10;
    int c = 20;

    switch ( a ) {
        case b:
            cout<<"Hurray B";
            break;
        case c:
            cout<<"Hurray C";
            break;
        default:
            cout<<"Wrong code";
            break;
    }
}
```

2.2 Iterative Constructs (Looping):

Iterative constructs or Loops enables a program with a cyclic flow of logic. Each statement which is written under the scope of a looping statement gets executed the number of times the iteration/looping

continues for.

In previous chapter we have seen few real life scenarios where a looping construct is needed for performing a particular set of tasks repeatedly for a number of times. Now we will go into details of all looping constructs available in C++.

In C++ there are three basic types of looping constructs available, they are :

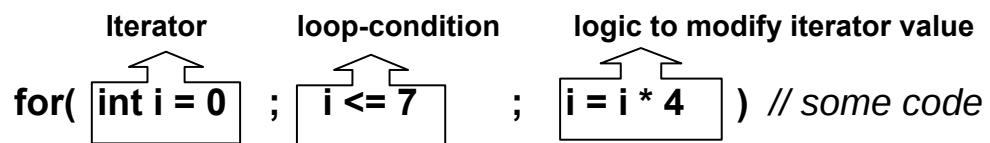
- while() loop // keyword while is used to loop
- do-while() loop // keywords do and while are used to loop
- for loop // keyword for is used to loop

It is very important to understand that a looping construct enables repetition of tasks under its scope based on a particular condition called as **loop-condition**. This loop-condition evaluates as true or false. A loop may be continued till a loop-condition is evaluated as true or false, based on a particular program situation.

All the above mentioned three loops have three parts common in them i.e.

- the looping variable (iterator)
- the loop-condition
- logic to change the value of iterator with each cycle/iteration

Let me show you distinguishably these three parts and then we will proceed to the syntax.


for(int i = 0 ; i <= 7 ; i = i * 4) // some code

Similarly , while and do-while also have these three significant parts as shown below :

```
int i = 0
while ( i <= 7 )
{
    // some code
    i = i * 4 ;
}
```

```
int i = 0;
do
{
    //some code
    i = i *4;
} while(i <= 7 ) ;
```

2.2.1 : while() loop construct

A while loop tests for its ending condition before performing its contents - even the first time. So if the ending condition is met when the while loop begins, the lines of instructions it contains will *never* be carried out.

Syntax :

```
while (<loop-condition>)
```

```
{  
...;  
...;  
}
```

A while continues iteration-cycle till its loop condition is evaluated as false. If the loop-condition is false for the first time iteration then loop will not execute even once.

Example :

```
int x = 0, sum = 0 ;  
cout<<"Input a natural number";  
cin>>x;  
while(x > 0 )  
{  
    sum = sum + x;  
    x -- ;  
}  
cout<<"The sum is :." << sum;
```

The above code snippet finds the sum of n natural number using while loop. The loop is executed till the x is greater than 0 , as soon x becomes 0 the loop is terminated. We observe that within the scope of the loop the value of x is decremented so that it approaches to its next previous value. Thus with each iteration the value of x is added to a variable sum and is decremented by 1.

Lets understand the above program with a Dry Run. Let us assume that user inputs a value 4 for x then at :

1st Iteration start

x is 4 , sum = 0 i.e. the loop-condition $4 > 0$ is evaluated as true hence it opens up the while block
==> $sum = 0 + 4 = 4$
x will be decremented by 1 , thus $x = 3$

2nd Iteration start

x is 3 , sum = 4 i.e. the loop-condition $3 > 0$ is evaluated as true hence it opens up the while block
==> $sum = 4 + 3 = 7$
x will be decremented by 1 , thus $x = 2$

3rd Iteration start

x is 2 , sum = 7 i.e. the loop-condition $2 > 0$ is evaluated as true hence it opens up the while block
==> $sum = 7 + 2 = 9$
x will be decremented by 1 , thus $x = 1$

4th Iteration start

x is 1 , sum = 9 i.e. the loop-condition $1 > 0$ is evaluated as true hence it opens up the while block
==> $sum = 9 + 1 = 10$
x will be decremented by 1 , thus $x = 0$

5th Iteration start

x is 0 , sum = 10 i.e. the loop-condition $0 > 0$ is evaluated as **false** because $0 == 0$ hence it locks up the while block and while() loop is terminated.

Thus coming out of while loop block the value of sum is printed as 10
loop executes for 4 times.

Workout yourself :

Consider the program given below :

```
void main( )
{
    int x = 5, m = 1 , p = 0;
    while( p <= 50 )
    {
        p = x * m ;
        m++;
    }
}
```

Now complete the following Iteration tracking table with details of each iteration, the first one is done for you :

Iteration	loop-condition	x	p	m
1 st Iteration	$0 \leq 50$ $\Rightarrow$ true	5	5	2

Check it out with your teacher.

Why my while loop is not getting executed ?

It is very important to understand here that in while loop the looping condition is evaluated at the beginning of the loop's scope i.e. prior to the entering into scope of loop. This type of checking is called “**Entry Control Checking**”, if the condition fails this checking it will not be allowed into the scope of the loop. That is why while() loops are often called as “**Entry Control Loop**”. You can imagine a similar checking situation that your tickets are being checked prior to your entry into a cinema hall to view a film. You will not be able to view the film at all if you don't have the tickets !!

Similarly if the loop-condition fails at the first time itself the statements under the scope of loop will not run even once. Observe such a code scenario below :

```
noOfPersons = 4;
noOfTickets = 3 ;
```

```

while( noOfTickets >= noOfPersons )
{
    cout<<"Welcome to Gangs of C++ pure"
}

```

The above loop will not execute at all because the loop-condition is failing at first time.

2.2.2 : do-while() loop construct

A do-while loop is identical to a while loop in every sense except that it is *guaranteed* to perform the instructions inside once before testing for the ending condition.

Syntax:

```

do
{
    // do something ;
    // do something ;
} while( <loop-condition> ) ;

```

Example: Consider the following code snippet to find factorial of a given number *n* :

```

int f = 1;
int n = 0;
cin>> n;

```

```

do
{
    f = f * n ;
    -- n ;
} while( n > 0 ) ;
cout << "The factorial of : " << n << " is " << f ;

```

Lets us analyze the execution of the above program : the factorial of any number *n* is evaluated as

$$\text{fact} = n * (n-1) * (n-2) * \dots * 1$$

This means that with each iteration the value of *n* is decremented by 1 and is multiplied with the previous value of *n* is stored cumulatively. The iteration-cycle stops when *n* is decremented upto a value equal to 1.

Let us conduct a Dry run on the above code snippet to understand the flow of logic for ***n = 4***:

Initially

$$f = 1, n = 4$$

1st Iteration-cycle:

$$f = 1 * 4 = 4, n = 3, \text{ condition } n > 0 \text{ evaluates as true hence loop is continued}$$

2nd Iteration-cycle :

$$f = 4 * 3 = 12, n = 2, \text{ condition } n > 0 \text{ evaluates as true hence loop is continued}$$

3rd Iteration-cycle :

$$f = 12 * 2 = 24, n = 1, \text{ condition } n > 0 \text{ evaluates as true hence loop is continued}$$

4th Iteration-cycle :

$f = 24 * 1 = 24$, $n = 0$, condition $n > 0$ evaluates as false , as $0 == 0$ hence loop is terminated and program flow comes out of the scope of the loop into program scope.

The output is printed on console as:

“The factorial of : 0 is 24”

Workout yourself :

Though the above factorial code snippet has executed nicely and produced correct and valid result of factorial of 4 i.e. 24 but while prompting output to user it wrongly says that “factorial of :0 is 24” . It must show that “factorial of 4 is 24”.

Find out the reason for the above misprinting and correct the program by modifying the code so that it produces expected output prompt to user.

Check your result with your teacher.

Why my do-while() loop is getting executed once even when the condition is not satisfied !

It is very important to understand here that in do-while loop the looping condition is evaluated at the end of the loop's scope i.e. after the last statement in the scope of loop. This type of checking is called **“Exit Control Checking”**, if the condition fails this checking it will not be allowed into the scope of the loop on next iteration-cycle. In the whole affair you are observing that the loop had executed at least once even when condition fails in 1st iteration-cycle. That is why do-while() loops are often called as **“Exit Control Loop”**. You can imagine a similar checking situation that your shopping bag items and purchase bill are being checked on your exit out of a shopping mall where you have been for shopping.

For example consider the program 2.3 ahead.

Sentinels :

Often a looping construct executes loop for a fixed number of times based on certain looping condition placed on the looping variable or the iterator, but sometimes the termination of loop is not fixed i.e. the termination of loop depends on some outside input. These types of loops where termination of loop is not fixed and depends on outside input are called **sentinels**.

For example : Consider the program given below :

program 2.3

// program to count number of adults and minors

```
#include<iostream.h>
```

```
void main( )
```

```
{
```

```
    int ad = 0 , m = 0;
```

```
    int age = 0;
```

```
    cout<< "Input age :";  
    cin>> age;
```

```

while(age > 0)
{
    if( age >= 18 )
        ad++ ;
    else
        m++;
    cout<< "Input age :";
    cin>> age;
}

cout<< "There are total " << ad <<" adults and" << m << "minors";
}

```

Can you tell how many times the loop in above question runs ? Your answer would be probably “No” , or better you will tell that since the termination of the loop depends upon the value of the variable age, it is not fixed that how many times the loop may run.

Yes this is the feature a sentinel has , here the value of variable age controls the execution cycle of the while () loop. If the value of the age inputted by the user before the start of the loop is inputted as 0 then the loop will not be executed at all because the condition fails on the start of the first cycle itself. Moreover it also not fixed that after how many cycles the user may input a 0 value to stop the loop , he/she may input after 3 cycles or 5 or may be 200 !

Workout yourself:

In the above program 2.3, remove the two shaded input line placed before the start of while () construct, and then execute the code to find whether you are getting same result as before or anything wrong. If incorrect result is produced then find the reason explain it to class friends.

2.2.3 : for(; ;) loop

A for loop has all of its three parts i.e the iterator variable , loop-condition and logic to continue loop kept intact at one place separated by semi colon (;) . This loop is also an **Entry Control Loop**, as condition is checked before entering into the scope of the loop. Let us analyze the syntax of a for looping:

Syntax:

```

for ( <variable(s) initialization> ; <looping-condition> ; <incrementing / decrementing> )
{
    // do something
    ...
}

```

The first part of a for loop is a place where all looping variables are declared and initialized in the same way we declare multiple variables of a single data type : e.g. int i = 0 , j = 2 , k = 9 ... ;

This part of the loop is executed only once at the start up of the loop and never it is executed again (how can you re-declare variables with the same name! in same scope). As per turbo c++ compiler the scope of all these variables declared in for loop is same as program-scope i.e. they can be accessed from any other scope in the main(), but the current ISO C++ has changed this making the scope of all these variable local to for loop only. You will follow the turbo c++ type scoping.

The second part of a for loop defines the looping condition using set of relational and logical operators and governs the number of times the loop would execute. This looping condition is checked before entering into loop, for e.g. $i \leq j+k$ etc.

The third part of the for loop defines how to change the value of iterative / looping variables with each cycle. This is very important as to execute the loop for a fixed number of times , for e.g. $i++$, $j = j+k$ etc

Let us now integrate all the above three parts to observe how a for loop works :

Example :

```
for(int i = 0 ; i<=3 ; i++)
{
    cout<<"Arun Kumar\n";
    cout<<"Kamal Kant Gupta\n";
    cout<<"Anil Kumar\n";
}
```

The above loop starts with declaring variable $i = 0$, then it proceeds to the looping-condition part to check whether condition is satisfied (evaluates as true) so that the flow could enter into the scope of the loop. i.e. $0 \leq 3$ which is true , the control flow is allowed to enter into the scope of the loop and all the statements (the 3 lines) gets executed one after another. After executing the last statement within the scope of for() the control proceeds to the third part of the loop where it increments the value of looping variable i by 1 , so that the variable is getting the next consecutive higher value i.e. $i = 1$. Hence the first Iteration of the for() loop is finished.

Now at the start of the next iteration the control flow directly proceeds to the second part of the for loop to evaluate looping-condition i.e. $1 \leq 3$ which is true , the flow of control then proceeds in the similar way as it did in the first iteration and this is repeated till the looping-condition becomes false i.e. when i will become equal to 4 .

Let us put all the things discussed above in the form of an iteration tracking table :

Iteration	i	Looping condition	Output
Starting of loop	0	$0 \leq 3 \rightarrow \text{true}$	Arun Kumar Kamal Kant Gupta Anil Kumar
End of 1 st Iteration	$i++ \rightarrow 1$	$1 \leq 3 \rightarrow \text{true}$	Same as above
End of 2 nd Iteration	$i++ \rightarrow 2$	$2 \leq 3 \rightarrow \text{true}$	Same as above

End of 3 rd Iteration	i++ → 3	3<=3 → true	Same as above
End of 4 th Iteration	i++ → 4	4<=3 → false	No entry into for block , loop is terminated

Observe the above table carefully to find that the loop has executed for 4 times, and the last value of the iterative variable on the termination of loop is 4.

Variations in for () loop :

All the three parts of a for loop are optional part i.e. they may or may not be present. Observe the valid variation in syntax of for () loop :

variation-1 :

```
int i = 0;
for( ; i<=5 ; i++ )
    cout<<"Hello to all";
```

In the above for () loop the first part has been left i.e. we have not declared and initialized the iterator in for () , though it has been declared outside the scope of for () .

variation-2 :

```
int i = 0 ;
for( ; ; i++)
{
    if( i<=5 )
        cout<<"Hello to all";
    else
        break;
}
```

In the above variation-2 the loop is not having first and second part defined for it, though the execution of the program remains same as version 1 since iterator has been declared and the condition for terminating the loop is implemented through a **if-else** construct inside the loop. You may observe that to terminate the loop we have used the **break statement**.

Whenever a break statement is found in a particular construct 's scope it immediately comes out of the current scope.

Variation-3 :

```
int i = 0 ;
for ( ; ; )
{
    if( i<=5 )
    {
        cout<<"Hello to all";
        i++;
    }
    else
        break;
}
```

The above variation-3 is implemented in a similar to that of the variation-3 but it has its incrementation statement defined within if() construct.

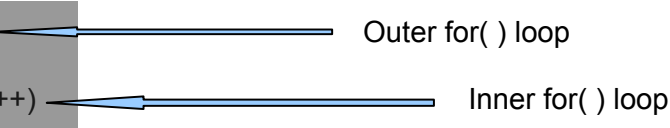
In all of the above variations you might have observed that **it is compulsory to put a semicolon (;) within for() even when statements before and after it are absent.**

2.3 : Nested Loops

Students in section 2.1.7 we have seen that any conditional construct can be nested within any other conditional construct. Similarly any looping construct can also be nested within any other looping construct and conditional constructs.

Let us look at the following example showing the nesting of a for() loop within the scope of another for() loop :

```
for(int i = 1 ; i<=2 ; i++)  
{  
    for( int j = 1 ; j<=3 ; j++)  
    {  
        cout<< i * j <<endl ;  
    }  
}
```



For each iteration of the outer for loop the inner for loop will iterate fully up to the last value of inner loop iterator. The situation can be understood more clearly as :

1st Outer Iteration
i= 1

1st Inner Iteration
j = 1 , output : 1 * 1 = 1

2nd Inner Iteration
j = 2 , output : 1 * 2 = 2

3rd Inner Iteration
j = 3 , output : 1 * 3 = 3

2nd Outer Iteration
i= 2

1st Inner Iteration
j = 1 , output : 2 * 1 = 2

2nd Inner Iteration
j = 2 , output : 2 * 2 = 4

3rd Inner Iteration
j = 3 , output : 2 * 3 = 6

You can observe that j is iterated from 1 to 3 every time i is iterated once.

Workout Yourself :

Write a program to print a multiplication table from 1 to 10 using concept of nested for loop

Let us summarize what we have learned till now :

- i) There are two basic types of Conditional constructs : if() construct and switch-case statement. The if() has four different types of variations simple , compound , complex and nested.
- ii) There are three basic types of Iterative Constructs : while() , do-while() , and for() . Out of these while() and for() have similar flow of execution whereas do-while() is bit different.
- iii) for() and while() are called as Entry Control Loop , whereas do-while() is called as exit-control loop.
- iv) Any of the Iterative construct could be nested within the scope of another iterative construct. A for() can be enclosed within scope of another for() , while() within a for() , a for() within a do-while() etc.
- v) while programming for real life situation we are often going to mix conditional and iterative flow of logic as and when required.

Check your progress:

1. Write C++ program to sum of following series:

i) $S = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6} + \dots + n\text{terms}$

ii) $S = 1 + \frac{1}{1^2} + \frac{1}{2^3} + \frac{1}{3^4} + \dots + n\text{terms}$

iii) $S = 1 - 2 + 3 - 5 + 6 - 7 + \dots n\text{ terms}$

iv) $S = 1 + \frac{1}{x!} + \frac{2^2}{(x-1)!} + \frac{3^3}{(x-2)!} + \dots \frac{n^n}{(x-n)!}$

2. Using while loop find the sum of digits of number if the inputted number is prime, if number is

then print the nearest prime number.

For example: if user inputs number = 17 then the output would be = $1 + 7 = 8$

if user inputs number = 20 then the output would be = 23 (next prime to 20)

3. You might be knowing DNA in Biology , it is a genetic molecule made of four basic types of nucleotides. The four nucleotides are given one letter abbreviations as shorthand for the four bases.

A is for adenine

G is for guanine

C is for cytosine

T is for thymine

When considering the structure of DNA it is made of two strands. Each of these strands are made of long chains of above nucleotides like :

AAGCTCAGAGCTATG 1st strand

TTCGAGTCTCGATAC 2nd strand

Each of the nucleotide of 1st strand pairs with nucleotide in other strand using bond. These bonding obeys the following rule:

I) A will always pair with T and vice versa

II) G will always pair with C and vice versa

as it can be observed in the two strands given above.

Write a program in C++ which allows user to input the nucleotide character sequence of 1st strand and print the probable sequence of the other strand. The user must provided opportunity to input a strand of any size, and only stops when user inputs an out of order character instead of A , G , C , & T.