



Key concepts

- **Decision making statements**
 - if statement
 - if ... else statement
 - Nested if
 - else if ladder
 - switch statement
 - Conditional operator
- **Iteration statements**
 - while statement
 - for statement
 - do... while statement

Control Statements

In the previous chapters we discussed some executable statements of C++ to perform operations such as input, output and assignment. We know how to write simple programs. The execution of these programs is sequential in nature. That is the statements constituting the program are executed one by one. In this chapter, we discuss C++ statements used for altering the default flow of execution. As we discussed in Chapter 3, selection, skipping or repeated execution of some statements may be required for solving problems. Usually this decision will be based on some condition(s). C++ provides statements to facilitate this requirement with the help of control statements. These statements are used for altering the normal flow of program execution. Control statements are classified into two: (i) decision making/selection statements and (ii) iteration statements. Let us discuss these statements, their syntax and mode of execution.

7.1 Decision making statements

At times it so happens that the computer may not execute all statements while solving problems. Some statements may be executed in a situation, while they may not be executed

in other situations. The computer has to take the required decision in this respect. For this, we have to provide appropriate conditions which will be evaluated by the computer. It will then take a decision on the basis of the result. The decision will be in the form of selecting a particular statement for execution or skipping some statements from being executed. The statements provided by C++ for the selected execution are called **decision making statements** or **selection statements**. **if** and **switch** are the two types of selection statements in C++.

7.1.1 if statement

The **if** statement is used to select a set of statements for execution based on a condition. In C++, conditions, otherwise known as test expressions are provided by relational or logical expressions. The syntax (general form) of **if** statement is as follows:

```
if (test expression)
{
    statement block;
}
```

Body of **if** statement that consists of the statement(s) to be executed when the condition is True

Here the `test expression` represents a condition which is either a relational expression or logical expression. If the test expression evaluates to True (non-zero value), a statement or a block of statements associated with **if** is executed. Otherwise, the control moves to the statement following the **if** construct. Figure 7.1 shows the execution mode of **if** statement. While using **if**, certain points are to be remembered.

- The test expression is always enclosed in parentheses.
- The expression may be a simple expression constituted by relational expression or a compound expression constituted by logical expression.
- The statement block may contain a single statement or multiple statements. If there is a single statement, then it is not mandatory to enclose it in curly braces `{ }`. If there are multiple statements, then they must be enclosed in curly braces.

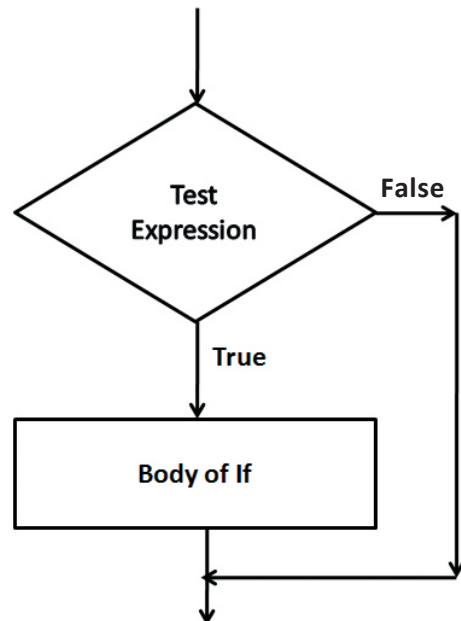


Fig. 7.1 : Working of if statement

Program 7.1 accepts the score of a student and displays the text “You have passed” only if he/she has passed. (assume that 18 is the minimum for pass).

Program 7.1: To display ‘You have Passed’ if score is 18 or more

```
#include <iostream>
using namespace std;
int main()
{
    int score ;
    cout << "Enter your score: ";
    cin >> score;
    if (score >= 18)
        cout << "You have Passed";
    return 0;
}
```

Body of **if**

The following is a sample output of program 7.1:

```
Enter your score : 25
You have passed
```

In Program 7.1, the score of a student is entered and stored in the variable `score`. The test expression compares the value of `score` with 18. The body of `if` will be executed only if the test expression evaluates to True. That means, when the score is greater than or equal to 18, the output “You have Passed” will be displayed on the screen. Otherwise, there will be no output.

Note that, the statement block associated with `if` is written after a tab space. We call it indentation. This is a style of coding which enhances the readability of the source code. Indentation helps the debugging process greatly. But it has no impact on the execution of the programme.

Consider the following C++ program segment. It checks whether a given character is an alphabet or a digit.

```
char ch;
cin >> ch;
if (ch >= 'a' && ch <= 'z')
    cout << "You entered an alphabet";
if (ch >= '0' && ch <= '9')
{
    cout << "\nYou entered a digit";
    cout << "\nIt is a decimal number";
}
```

Logical expression is evaluated

Only a single statement. No need of braces { }

More than one statement; Must be enclosed in braces { }

7.1.2 if...else statement

Consider the if statement in Program 7.1:

```
if (score >= 18)
    cout << "You have Passed";
```

Here, the output is obtained only if the score is greater than or equal to 18. What will happen if the score entered is less than 18? It is clear that there will be no output. Actually we don't have the option of selecting another set of statements if the test expression evaluates to False. If we want to execute some actions when the condition becomes False, we introduce another form of if statement, if...else. The syntax is:

```
if (test expression)
{
    statement block 1;
}
else
{
    statement block 2;
}
```

Here, if the test expression evaluates to True, only the statement block 1 is executed. If the test expression evaluates to False, statement block 2 is executed. The flowchart shown in Figure 7.2 explains the execution of if...else statement.

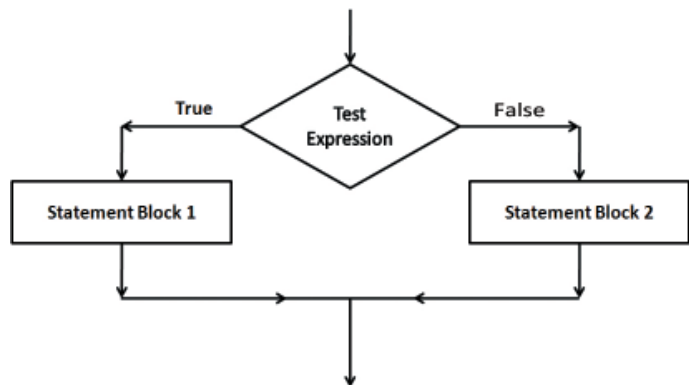


Fig 7.2: Flowchart of if-else

The following code segment illustrates the working of if...else statement.

```
if (score >= 18)
    cout << "Passed";
else
    cout << "Failed";
```

This statement is executed only when score is 18 or more (Test expression returns True)

This statement is executed only when score is less than 18 (Test expression returns False)

Let us write a program to input the heights of two students and find the taller.

Program 7.2: To find the taller student by comparing their heights

```
#include <iostream>
using namespace std;
int main()
{
    int ht1, ht2;
    cout << "Enter heights of the two students: ";
    cin >> ht1 >> ht2;
    if (ht1 > ht2)    //decision making based on condition
        cout << "Student with height "<<ht1<<" is taller";
    else
        cout << "Student with height "<<ht2<<" is taller";
    return 0;
}
```

When program 7.2 is executed, one of the output statements will be displayed. The selection depends upon the relational expression $ht1 > ht2$. The following are sample outputs:

Output 1:

```
Enter heights of the two students: 170    165
Student with height 170 is taller
```

Output 2:

```
Enter heights of the two students: 160    171
Student with height 171 is taller
```

In the first output, we input 170 for $ht1$ and 165 for $ht2$. So, the test expression ($ht1 > ht2$) is evaluated to True and hence the statement block of `if` is selected and executed. In the second output, we input 160 for $ht1$ and 171 for $ht2$. The test expression, ($ht1 > ht2$) is evaluated and found False. Hence the statement block of `else` is selected and executed.

In `if...else` statement, either the code associated with `if` (statement block 1) or the code associated with `else` (statement block 2) is executed.

Let us see another program that uses an arithmetic expression as one of the operands in the test expression. Program 7.3 uses this concept to check whether an input number is even or odd.

Program 7.3: To check whether a given number is even or odd

```
#include <iostream>
using namespace std;
int main()
{
    int num;
    cout<<"\nEnter the number: ";
    cin >> num;
    if (num%2 == 0)
        cout << "The given number is Even";
    else
        cout << "The given number is Odd";
    return 0;
}
```

Some sample outputs of Program 7.3 are shown below:

Output 1:

```
Enter the number: 7
The given number is Odd
```

Output 2:

```
Enter the number: 10
The given number is Even
```

In this program, the expression $(num\%2)$ finds the remainder when `num` is divided by 2 and compares it with the value 0. If they are equal, the `if` block is executed, otherwise the `else` block is executed.



Let us do

1. Write a program to check whether a given number is a non-zero integer number and is positive or negative.
2. Write a program to input enter a single character for sex and print out the gender.
If the input character is 'M' display "Male". If the input is 'F' display "Female"
3. Write a program to input your age and check whether you are eligible to cast vote (the eligibility is 18 years and above).

7.1.3 Nested if

In some situations there may arise the need to take a decision within `if` block. When we write an `if` statement inside another `if` block, it is called nesting. Nested means one inside another.

Consider the following program segment:

```

if (score >= 60)
{
    if(age >= 18)
        cout<<"You are selected for the course!";
}

```

outer if

inner if

In this code fragment, if the value of `score` is greater than or equal to 60, the flow of control enters the statement block of `outer if`. Then the test expression of the `inner if` is evaluated (i.e. whether the value of `age` is greater than or equal to 18). If it is evaluated to `True`, the code displays the message, "You are selected for the course!". Then the program continues to execute the statement following the `outer if` statement. An `if` statement, inside another `if` statement is termed as a **nested if** statement. The following is an expanded form of `nested if`.

```

if (test expression 1)
{
    if (test expression 2)
        statement 1;
    else
        statement 2;
}
else
{
    body of else ;
}

```

It will be executed if both the test expressions are True.

It will be executed if test expression 1 is True, but test expression 2 is False.

It will be executed if test expression 1 is False. The test expression 2 is not evaluated.

The important point to remember about `nested if` is that an `else` statement always refers to the nearest `if` statement within the same block. Let us discuss this case with an example. Consider the following program segment:

```

cout<<"Enter your scores in Computer Science exam: ";
cin>>score;
if (score >= 18)
    cout<<"You have passed";
    if(score >= 54)
        cout<<" with A+ grade !";
else
    cout<<"\nYou have failed";

```

If we input the value 45 for score, the output will be as follows:

```
You have passed
You have failed
```

We know that this is logically not correct. Though the indentation of the code is proper, that doesn't matter in execution. The second `if` statement will not be considered as nested `if`, rather it is counted as an independent `if` with an `else` block. So, when the first `if` statement is executed, the `if` block is selected for execution since the test expression is evaluated to `True`. It causes the first line in the output. After that, while considering the second `if` statement, the test expression is evaluated to `False` and hence the second line in the output is obtained. So to get the correct output, the code should be modified as follows:

```
cout<<"Enter your scores in Computer Science exam: ";
cin>>score;
if (score >= 18)
{
    cout<<"You have passed";
    if(score >= 54)
        cout<<" with A+ grade !";
}
else
    cout<<"\nYou have failed";
```

Nesting is enforced by putting a pair of braces

It is now associated with the outer if

If we input the same value 45 as in the case of previous example, the output will be as follows:

```
You have passed
```

Program 7.4 uses nested `if` to find the largest among three given numbers. In this program, `if` statement is used in both the `if` block and `else` block.

Program 7.4: To find the largest among three numbers

```
#include <iostream>
using namespace std;
int main()
{
    int x, y, z;
    cout << "Enter three different numbers: ";
    cin >> x >> y >> z ;
```



```
if (x > y)
{
    if (x > z)
        cout << "The largest number is: " << x;
    else
        cout << "The largest number is: " << z;
}
else
{
    if (y > z)
        cout << "The largest number is: " << y;
    else
        cout << "The largest number is: " << z;
}
return 0;
}
```

A sample output of program 7.4 is given below:

```
Enter three different numbers: 6    2    7
The largest number is: 7
```

As per the input given above, the test expression ($x > y$) in the outer `if` evaluated to True and hence the control enters the inner `if`. Here the test expression ($x > z$) is evaluated to False and so its `else` block is executed. Thus the value of `z` is displayed as the output.



Check yourself

1. Write a program to input an integer and check whether it is positive, negative or zero.
2. Write a program to input three numbers and print the smallest one.

7.1.4 The `else if` ladder

There are situations where an `if` statement is used within an `else` block. It is used in programs when multiple branching is required. Different conditions will be given and each condition will decide which statement is to be selected for execution. A common programming construct based on `if` statement is the `else if` ladder, also referred to as the `else if` staircase because of its appearance. It is also known as `if...else if` statement.

The general form of `else if` ladder is:

```
if (test expression 1)
    statement block 1;
    else if (test expression 2)
        statement block 2;
        else if (test expression 3)
            statement block 3;
            .....
            else
                statement block n;
```

At first, the test expression 1 is evaluated and if it is True, the statement block 1 is executed and the control comes out of the ladder. That means, the rest of the ladder is bypassed. If test expression 1 evaluates to False, then the test expression 2 is evaluated and so on. If any one of the test expressions evaluates to True, the corresponding statement block is executed and control comes out of the ladder. If all the test expressions are evaluated to False, the statement block n after the final `else` is executed. Observe the indentation provided in the syntax and follow this style to use `else if` ladder.

Let us illustrate the working of the `else if` ladder by writing a program to find the grade of a student in a subject when the score out of 100 is given. The grade is found out by the criteria given in the following table:

Score	Grade
80 or more	A
From 60 To 79	B
From 40 To 59	C
From 30 To 39	D
Below 30	E

Program 7.5: To find the grade of a student for a given score

```
#include <iostream>
using namespace std;
int main()
{
    int score;
    cout << "Enter the score: ";
```

```
cin >> score;
if (score >= 80)
    cout<<"A Grade";
    else if (score >= 60)
        cout<<"B Grade ";
        else if (score >= 40)
            cout<<"C grade";
            else if (score >= 30)
                cout<<"D grade";
                else
                    cout<<"E Grade";
}
```

The following are the sample outputs of program 7.5:

Output 1:

```
Enter the score: 75
B Grade
```

Output 2:

```
Enter the score: 25
E Grade
```

In program 7.5, initially the test expression `score>=80` is evaluated. Since the input is 75 in Output 1, the test expression is evaluated to False, and the next test expression `score>=60` is evaluated. Here it is True, and hence "B Grade" is displayed and the remaining part of the `else if` ladder is bypassed. But in the case of Output 2, all the test expressions are evaluated to False, and so the final `else` block is executed which makes "E Grade" as the output.

Let us write a program to check whether the given year is a leap year or not. The input value should be checked to know whether it is century year (year divisible by 100). If it is a century year, it becomes a leap year only if it is divisible by 400 also. If the input value is not a century year, then we have to check whether it is divisible by 4. If it is divisible, the given year is a leap year, otherwise it is not a leap year.

Program 7.6: To check whether the given year is leap year or not

```
#include <iostream>
using namespace std;
int main()
{
    int year ;
    cout << "Enter year (in 4-digits): ";
    cin >> year;
```

```

if (year % 100 == 0) // Checks for century year
{
    if (year % 400 == 0)
        cout << "Leap Year\n" ;
    else
        cout<< "Not a leap year\n" ;
}
else if (year % 4 == 0)
    cout << "Leap Year\n" ;
else
    cout<< "Not a leap year\n" ;
}

```

A century year is leap year only if it is divisible by 400

Non - century year is leap year only if it is divisible by 4

Let us see some sample outputs of Program 7.6:

Output 1:

```

Enter year (in 4-digits): 2000
Leap Year

```

Output 2:

```

Enter year (in 4-digits): 2014
Not a leap Year

```

Output 3:

```

Enter year (in 4-digits): 2100
Not a leap Year

```

Output 4:

```

Enter year (in 4-digits): 2004
Leap Year

```

We write one more program to illustrate the use of `else if` ladder. Program 7.7 allows to input a number between 1 and 7 to denote the day of a week and displays the name of the corresponding day. The input 1 will give you "Sunday" as output, 2 will give "Monday" and so on. If the input is outside the range 1 to 7, the output will be "Wrong input".

Program 7.7: To display the name of the day for a given day number

```

#include <iostream>
using namespace std;
int main()
{
    int day;
    cout << "Enter day number (1-7): ";
    cin >> day;
    if (day == 1)
        cout<<"Sunday" ;
        else if (day == 2)

```

```

        cout<<"Monday" ;
        else if (day == 3)
            cout<<"Tuesday" ;
            else if (day == 4)
                cout<<"Wednesday" ;
                else if (day == 5)
                    cout<<"Thursday" ;
                    else if (day == 6)
                        cout<<"Friday" ;
                        else if (day == 7)
                            cout<<"Saturday" ;
                            else
                                cout<<"Wrong input" ;

    return 0;
}

```

The following are some sample outputs of Program 7.7:

Output 1:

```

Enter day number (1-7): 5
Thursday

```

Output 2:

```

Enter day number (1-7): 9
Wrong input

```



Check yourself

1. Write a program to input an integer number and check whether it is positive, negative or zero using `if...else if` statement.
2. Write a program to input a character (a, b, c or d) and print as follows:
a - "abacus", b - "boolean", c - "computer" and d - "debugging"
3. Write a program to input a character and print whether it is an alphabet, digit or any other character.

7.1.5 switch statement

We have seen the concept of multiple branching with the help of `else if` ladder. Some of these programs can be written using another construct of C++ known as `switch` statement. This selection statement successively tests the value of a variable or an expression against a list of integer or character constants.

The syntax of switch statement is given below:

```
switch (expression)
{
    case constant_1    : statement block 1;
                       break;
    case constant_2    : statement block 2;
                       break;
    case constant_3    : statement block 3;
                       break;
                       :
                       :
    case constant_n-1  : statement block n-1;
                       break;
    default            : statement block n;
}

```

In the syntax **switch**, **case**, **break** and **default** are keywords. The expression is evaluated to get an integer or character constant and it is matched against the constants specified in the `case` statements. When a match is found, the statement block associated with that `case` is executed until the `break` statement or the end of `switch` statement is reached. If no match is found, the statements in the `default` block get executed. The `default` statement is optional and if it is missing, no action takes place when all matches fail. The `break` statement, used inside `switch`, is one of the jump statements in C++. When a `break` statement is encountered, the program control goes to the statements following the `switch` statement.

Program 7.7 can be written using `switch` statement. It enhances the readability and effectiveness of the code. Observe the modification in Program 7.8.

Program 7.8: To display the day of a week using switch statement

```
#include <iostream>
using namespace std;
int main()
{
    int day;
    cout << "Enter a number between 1 and 7: ";
    cin >> day;
    switch (day)
    {
        case 1:    cout << "Sunday";
                  break;
        case 2:    cout << "Monday";
                  break;
        case 3:    cout << "Tuesday";
                  break;
    }
}

```

```

        case 4:    cout << "Wednesday";
                  break;
        case 5:    cout << "Thursday";
                  break;
        case 6:    cout << "Friday";
                  break;
        case 7:    cout << "Saturday";
                  break;
        default:   cout << "Wrong input";
    }
}

```

The output of Program 7.8 will be the same as in Program 7.7. The following are some samples:

Output 1:

```

Enter a number between 1 and 7: 5
Thursday

```

Output 2:

```

Enter a number between 1 and 7: 8
Wrong input

```

In Program 7.8, value of the variable `day` is compared against the constants specified in the `case` statements. When a match is found, the output statement associated with that `case` is executed. If we input the value 5 for the variable `day`, then the match occurs for the fifth `case` statement and the statement `cout << "Thursday";` is executed. If the input is 8 then no match occurs and hence the `default` block is executed.

Can you predict the output of Program 7.8, if all the `break` statements are omitted? The value returned by `day` is compared with the `case` constants. When the first match is found, the associated statements will be executed and the following statements will also be executed irrespective of the remaining constants. There are situations where we omit the `break` statement purposefully. If the statements associated with all the `case` in a `switch` are the same we only need to write the statement against the last `case`. Program 7.9 illustrates this concept.

Program 7.9: To check whether the given character is a vowel or not

```

#include <iostream>
using namespace std;
int main()
{
    char ch;
    cout<<"Enter the character to check: ";
    cin>>ch;

```

```

switch(ch)
{
    case 'A':
    case 'a':
    case 'E':
    case 'e':
    case 'I':
    case 'i':
    case 'O':
    case 'o':
    case 'U':
    case 'u': cout<<"Given character is a vowel";
              break;
    default : cout<<"Given character is not a vowel";
}
return 0;
}

```

Some of the outputs given by Program 7.9 are shown below:

Output 1:

```

Enter the character to check: E
Given character is a vowel

```

Output 2:

```

Enter the character to check: k
Given character is not a vowel

```

Suitability and requirements for using **switch**

Though **switch** statement and **else if** ladder cause multiple branching, they do not work in the same fashion. In C++ all **switch** statements can be replaced by **else if** ladders, but all **else if** ladders cannot be substituted by **switch**. The following are the requirements to implement a multi branching using **switch** statement:

- Conditions involve only equality checking. In other cases, it should be converted into equality expression.
- The first operand in all the equality expressions should be the same variable or expression.
- The second operand in these expressions should be integer or character constants.

Among the programs we discussed so far in this chapter, only the branching in Programs 7.3 and 7.7 can be replaced by **switch**. In Program 7.5, we can use **switch** if we modify the test expressions as `score/10==10`, `score/10==9`, `score/10==8`, and so on.

That is the following code fragment may be used instead of `else if` ladder.

```
switch(score/10)
{
    case 10 :
    case 9  : case 8 : cout<< "A Grade"; break;
    case 7  : case 6 : cout<< "B Grade"; break;
    case 5  : case 4 : cout<< "C Grade"; break;
    case 3  :          cout<< "D Grade"; break;
    default :          cout<< "E Grade";
}
```

Let us have comparison between `switch` and `else if` ladder as indicated in table 7.1

switch statement	else if ladder
• Permits multiple branching.	• Permits multiple branching.
• Evaluates conditions with equality operator only.	• Evaluate any relational or logical expression.
• Case constant must be an integer or a character type value.	• Condition may include range of values and floating point constants.
• When no match is found, default statement is executed.	• When no expression evaluates to True, else block is executed.
• <code>break</code> statement is required for exit from the <code>switch</code> statement.	• Program control automatically goes out after the completion of a block.
• More efficient when the same variable or expression is compared against a set of values for equality.	• More flexible and versatile compared to <code>switch</code> .

Table 7.1: Comparison between switch and else if ladder

7.1.6 The conditional operator (?:)

As we mentioned in Chapter 5, C++ has a ternary operator. It is the conditional operator (?:) consisting of symbols ? and : (a question mark and a colon). It requires three operands to operate upon. It can be used as an alternative to `if...else` statement. Its general form is:

```
Test expression ? True_case code : False_case code;
```

The `Test expression` can be any relational or logical expression and `True_case code` and `False_case code` can be constants, variables, expressions or statement.

The operation performed by this operator is shown below with the help of an `if` statement.

```
if (Test expression)
{
    True_case code;
}
else
{
    False_case code;
}
```

The conditional operator works in the same way as `if...else` works. It evaluates the `Test expression` and if it is `True`, the `True_case code` is executed. Otherwise, `False_case code` is executed. Program 7.10 illustrates the working of conditional operator.

Program 7.10: To find the larger number using conditional operator

```
#include <iostream>
using namespace std;
int main()
{
    int n1, n2;
    cout << "Enter two numbers: ";
    cin >> n1 >> n2 ;
    (n1>n2)? cout<<n1<<" is larger" : cout<<n2<<" is larger";
}
```

The last statement of the program may be called conditional statement as it uses conditional operator. This statement may be replaced by the following code segment:

```
int big = (n1>n2) ? n1 : n2;
cout << big << " is larger";
```

If the test expression evaluates to `True`, the value of `n1` will be assigned to `big`, otherwise that of `n2`. Here conditional operator is used to construct a conditional expression. The value returned by this expression will be assigned to `big`.



Check yourself

1. Write a program to input a number in the range 1 to 12 and display the corresponding month of the year. (January for the value 1, February for 2, etc.)
2. What is the significance of `break` statement within a `switch` statement?
3. Rewrite the following statement using `if...else` statement

```
result= mark>30 ? 'p' : ' f';
```

7.2 Iteration statements

In Chapter 3, we discussed some problems for which the solution contains some tasks that were executed repeatedly. While writing programs, we use some specific constructs of the language to perform the repeated execution of a set of one or more statements. Such constructs are called **iteration statements** or **looping statements**. In C++, we have three iteration statements and all of them allow a set of instructions to be executed repeatedly when a condition is True.

We use the concept of loop in everyday life. Let us consider a situation. Suppose your class teacher has announced a gift for every student securing A+ grade in an examination. You are assigned the duty of wrapping the gifts. The teacher has explained the procedure for wrapping the gifts as follows:

- Step 1 : Take the gift
- Step 2 : Cut the wrapping paper
- Step 3 : Wrap the gift
- Step 4 : Tie the cover with a ribbon
- Step 5 : Fill up a name card and paste it on the gift pack

If there are 30 students with A+ grade in the examination, you have to repeat the same procedure 30 times. To repeat the wrapping process 30 times, the instructions can be restructured in the following way.

```
Repeat the following steps 30 times
{
    Take the next gift
    Cut the wrapping paper
    Wrap the gift
    Tie the cover with a ribbon
    Fill up a name card and paste on the gift pack
}
```

Let us take another example. Suppose we want to find the class average of scores obtained in Computer Applications. The following steps are to be performed:

```
Initially Total_Score has no value
Repeat the following steps starting from the first student till the last
{
    Add Score of the student to the Total_Score
    Take the Score of the next student
}
Average = Total_Score / No. of students in the class
```

In both the examples, we perform certain steps for a number of times. We use a counter to know how many times the process is executed. The value of this counter decides whether to continue the execution or not. Since loops work on the basis of such

conditions, a variable like the counter will be used to construct a loop. This variable is generally known as **loop control variable** because it actually controls the execution of the loop. In Chapter 3, we discussed four elements of a loop. Let us refresh them:

1. **Initialisation:** Before entering a loop, its control variable must be initialised. During initialisation, the loop control variable gets its first value. The initialisation statement is executed only once, at the beginning of the loop.
2. **Test expression:** It is a relational or logical expression whose value is either True or False. It decides whether the loop-body will be executed or not. If the test expression evaluates to True, the loop-body gets executed, otherwise it will not be executed.
3. **Update statement:** The update statement modifies the loop control variable by changing its value. The update statement is executed before the next iteration.
4. **Body of the loop:** The statements that need to be executed repeatedly constitute the body of the loop. It may be a simple statement or a compound statement.

We learnt in Chapter 3 that loops are generally classified into entry-controlled loops and exit-controlled loops. C++ provides three loop statements: **while** loop, **for** loop and **do . . . while** loop. Let us discuss the working of each one in detail.

7.2.1 while statement

The **while** loop is an entry-controlled loop. The condition is checked first and if it is found True the body of the loop will be executed. That is the body will be executed as long as the condition is True. The syntax of while loop is:

```
initialisation of loop control variable;
while(test expression)
{
    body of the loop;
    updation of loop control variable;
}
```

Here, the test expression defines the condition which controls the loop. The body of the loop may be a single statement or a compound statement or without any statement. The body is the set of statements for repeated execution. Update expression refers to a statement that changes the value of the loop control variable. In a **while** loop, a loop control variable should be initialised before the loop begins and it should be updated inside the body of the loop. The flowchart in Figure 7.3 illustrates the working of a **while** loop.

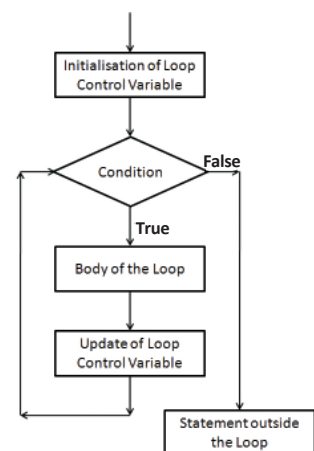


Fig 7.3: Working of while loop

The initialisation of the loop control variable takes place first. Then the `test` expression is evaluated. If it returns `True` the body of the loop is executed. That is why `while` loop is called an **entry controlled loop**. Along with the loop body, the loop control variable is updated. After completing the execution of the loop body, test expression is again evaluated. The process is continued as long as the condition is `True`. Now, let us consider a code segment to illustrate the execution of `while` loop.

```
int k=1;
while(k<=3 )
{
    cout << k << '\t';
    ++k;
}
```

Annotations for the code above:

- Initialization before loop begins (points to `int k=1;`)
- Test expression (points to `while(k<=3)`)
- Body of loop (points to `cout << k << '\t';` and `++k;`)
- Update inside the loop body (points to `++k;`)

In this code segment, the value 1 is assigned to the variable `k` (loop control variable) at first. Then the test expression `k<=3` is evaluated. Since it is `True`, the body of the loop is executed. That is the value of `k` is printed as 1 on the screen. After that the update statement `++k` is executed and the value of `k` becomes 2. The condition `k<=3` is checked again and found to be `True`. Program control enters the body of the loop and prints the value of `k` as 2 on the screen. Again the update statement is executed and the value of `k` is changed to 3. Since the condition is still `True`, body is executed and 3 is displayed on the screen. The value of `k` is again updated to 4 and now the test expression is evaluated to be `False`. The control comes out of the loop and executes the next statement after the `while` loop. In short, the output of the code will be:

1 2 3

Imagine what will happen if the initial value of `k` is 5? The test expression is evaluated to `False` in the first evaluation and the loop body will not be executed. This clearly shows that `while` loop controls the entry into the body of the loop.

Let us see a program that uses `while` loop to print the first 10 natural numbers.

Program 7.11: To print the first 10 natural numbers

```
#include <iostream>
using namespace std;
int main()
{
    int n = 1;
    while(n <= 10)
    {
        cout << n << " ";
        ++n;
    }
}
```

Annotations for the code above:

- Initialisation of loop variable (points to `int n = 1;`)
- Test expression (points to `while(n <= 10)`)
- Body of loop (points to `cout << n << " ";` and `++n;`)
- Updating loop variable (points to `++n;`)

The output of Program 7.11 will be as follows:

1 2 3 4 5 6 7 8 9 10

Program 7.12 uses `while` loop to find the sum of even numbers upto 20. This program shows that the loop control variable can be updated using any operation.

Program 7.12: To find the sum of even numbers upto 20

```
#include <iostream>
using namespace std;
int main()
{
    int i, sum = 0;
    i = 2;
    while( i<= 20)
    {
        sum = sum + i;
        i = i + 2;
    }
    cout<<"\nsum of even numbers up to 20 is: "<<sum;
    return 0;
}
```

Updating of loop control variable by adding 2 to the current value

The output of Program 7.12 is given below:

sum of even numbers up to 20 is: 110



Let us do

Modify Program 7.11 to display all odd numbers between 100 and 200.

Modify Program 7.12 to find the sum of the first N natural numbers.



If we put a semi colon (;) after the test expression of while statement, there will not be any syntax error. But the statements within the following pair of braces will not be considered as loop body. The worst situation is that, if the test expression is evaluated to be True, neither the code after the while loop will be executed nor the program will be terminated. It is a case of infinite loop.

7.2.2 for statement

The `for` loop is also an entry-controlled loop in C++. All the three loop elements (initialisation, test expression and update statement) are placed together in `for` statement. So it makes the program compact.

The syntax is:

```
for (initialisation; test expression; update statement)
{
    body-of-the-loop;
}
```

The execution of **for** loop is the same as that of **while** loop. The flowchart used for **while** can explain the working of **for** loop. Since the three elements come together this statement is more in situations where counting is involved. So the flowchart given in Figure 7.4 is commonly used to show the execution of **for** statement. At first, the

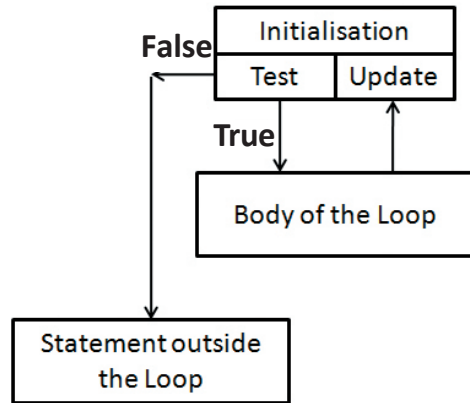


Fig 7.4: Execution of **for** loop

initialisation takes place and then the **test** expression is evaluated. If its result is **True**, **body-of-the-loop** is executed, otherwise the program control goes out of the **for** loop. After the execution of the loop body, **update** expression is executed and again **test** expression is evaluated. These three steps (**test**, **body**, **update**) are continued until the **test** expression is evaluated to **False**.

The loop segment used in Program 7.11 can be replaced with a **for** loop as follows:

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



Let us do

*The steps 1 and 2 in the execution sequence of the **for** loop just mentioned before are given below. Write down the remaining steps.*

Step 1: $N = 1$, Condition is True, 1 is displayed, N becomes 2

Step 2: Condition is True, 2 is displayed, N becomes 3

Step 3:

This code is executed in the same way as in the case of **while** loop.

Let us write a program using **for** loop to find the factorial of a number. Factorial of a number, say **N**, represented by **N!** is the product of the first **N** natural numbers. For example, factorial of 5 (**5!**) is calculated by $1 \times 2 \times 3 \times 4 \times 5 = 120$.

Program 7.13: To find the factorial of a number using for loop

```
#include <iostream>
using namespace std;
int main()
{
    int n, i, fact=1;
    cout<<"Enter number to find the factorial: ";
    cin>>n;
    for (i=1; i<=n; ++i)
        fact = fact * i ;
    cout << "Factorial of " << n << " is " << fact;
}
```

Initialisation; Test Expression; Updation

Loop body

The following is a sample output of program 7.13

```
Enter number to find the factorial: 6
Factorial of 6 is 720
```

Another program is given below which gives the class average of scores obtained in Computer Applications. Program 7.14 accepts the value for **n** as the number of students, then reads the score of each student and prints the average score.

Program 7.14: To find the average score of n students

```
#include <iostream>
using namespace std;
int main()
{
    int i, sum, score, n;
    float avg;
    cout << "How many students? ";
    cin >> n ;
    for(i=1, sum=0; i<=n; ++i)
    {
        cout << "Enter the score of student " << i <<": ";
        cin >> score;
        sum = sum + score;
    }
    avg = (float)sum / n;
    cout << "Class Average is: " << avg;
}
```

Initialisation contains two expressions

Explicit type conversion

The following is a sample output of Program 7.14 for 5 students

```
How many students? 5
Enter the score of student 1: 45
Enter the score of student 2: 50
Enter the score of student 3: 52
Enter the score of student 4: 34
Enter the score of student 5: 55
Class Average is: 47.2
```




While using `for` loops certain points are to be noted. The four code segments explain these special cases. Assume that all the variables used in the codes are declared with `int` data type.

Code segment 1:

```
for (n=1; n<5; n++);
    cout<<n;
```

A semicolon (;) appears after the parentheses of `for` statement. It is not a syntax error. Can you predict the output? If it is 5, you are correct. This loop has no body. But its process will be completed as usual. The initialisation assigns 1 to `n`, condition is evaluated to `True`, since there is no body update taking place and the process continues till `n` becomes 5. At that point, condition is evaluated to be `False` and the program control comes out of the loop. The execution of the output statement displays 5 on the screen

Code segment 2:

```
for (n=1; n<5; )
    cout<<n;
```

In this code, update expression is not present. It does not make any syntax error in the code. But on execution, the loop will never be terminated. The number 1 will be displayed infinitely. We call this an infinite loop.

Code segment 3:

```
for (; n<5; n++)
    cout<<n;
```

The output of this code cannot be predicted. Since there is no initialisation, the control variable `n` gets some integer value. If it is smaller than 5, the body will be executed until the condition becomes `False`. If the default value of `n` is greater than or equal to 5, the loop will be terminated without executing the loop body.

Code segment 4:

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

The test expression is missing in this code. `C++` takes this absence as `True` and obviously the loop becomes an infinite loop. The four code segments given reveal that all the elements of a `for` loop are optional. But this is not the case for `while` and `do...while` statements. Test expression is compulsory for these two loops. Other elements are optional, but be cautious about the output.

In Program 7.14, the initialisation contains two expressions `i=1` and `sum=0` separated by comma (,). The initialisation part may contain more than one expression, but they should be separated by comma. Both the variables `i` and `sum` get their first values 1 and 0 respectively. Then, the test expression `i<=n` is evaluated to be True and body of the loop is executed. After the execution of the body of the loop the update expression `++i` is executed. Again the test expression `i<=n` is evaluated, and body of the loop is executed since the condition is True. This process continues till the test expression returns False. It has occurred in the sample output when the value of `i` becomes 6.



Let us do

Write a program to display the multiplication table of a given number. Assume that the number will be the input to the variable `n`. The body of the loop is given below:

```
cout<<i<<" x "<<n<<" = "<< i * n <<"\n";
Give the output also.
```

7.2.3 do...while statement

In the case of `for` loop and `while` loop, the test expression is evaluated before executing the body of the loop. If the test expression evaluates to False for the first time itself, the body is never executed. But in some situations, it is necessary to execute the loop body at least once, without considering the result of the test expression. In that case the `do...while` loop is the best choice. Its syntax is:

```
initialisation of loop control variable;
do
{
    body of the loop;
    Updation of loop control variable;
} while(test expression);
```

Figure 7.5 shows the order of execution of this loop. Here, the test expression is evaluated only after executing body of the loop. So `do...while` loop is an **exit controlled loop**. If the test expression evaluates to False, the loop will be terminated. Otherwise, the execution process will be continued. It means that in `do...while` loop the body will be executed at least once irrespective of the result of the condition.

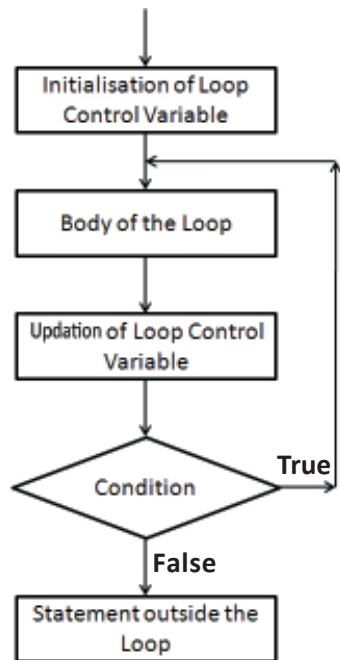


Fig. 7.5 : Flowchart of do...while loop

Let us consider the following program segment to illustrate the execution of a `do...while` loop.

```

int k=1;
do
{
    cout << k << '\t';
    ++k;
} while(k<=3);

```

Initialisation before the loop

Body of loop

Updation inside the loop body

Test expression at the end

At first, the value 1 is assigned to the variable **k**. Then body of the loop is executed and the value of **k** is printed as 1. After that the **k** is incremented by 1 (now $k=2$). Then it checks the condition $k \leq 3$. Since it is found True the body of the loop is executed to print the value of **k**, i.e. 2 on the screen. Again the updation process is carried out, which makes value of **k** as 3 and the condition $k \leq 3$ is checked again. As it is True, the body of the loop is executed to print the value 3. The variable **k** is again updated to 4 and now the condition is evaluated to be False. It causes the program control to come out of the loop and executes the next statement after the loop body. Thus the output of the code will be:

1 2 3




Now let us see how this loop differs from the other two. Imagine that the initial value of **k** is 5. What will happen? The body of the loop is executed and the value of **k** will be printed on the screen as 5. After that the variable **k** will be updated by incrementing it by 1 and **k** becomes 6. On checking the condition $k \leq 3$, the test expression is evaluated to False and the control comes out of the loop. This clearly shows that in `do...while` loop there is no restriction to enter the loop body for the first time. So if we want the body to be executed based on the True value of the condition, use `while` or `for` loops.

Let us see an interactive program in the sense that some part of the code will be executed on user's choice. The simplest form of such programs provides facility to accept user's response for executing a code segment in the program repeatedly. Program 7.15 illustrates the use of `do...while` loop to write an interactive program to find the area of rectangles by accepting the length and breadth of each rectangle from the user.

Program 7.15: To find the area of rectangles

```
#include <iostream>
using namespace std;
int main()
{
    float length, breadth, area;
    char ch;
    do
    {
        cout << "Enter length and breadth: ";
        cin >> length >> breadth;
        area = length * breadth;
        cout << "Area = " << area;
        cout << "Any more rectangle (Y/N)? ";
        cin >> ch;
    } while (ch == 'Y' || ch == 'y');
    return 0;
}
```

A sample output of Program 7.15 is given below:

```
Enter length and breadth: 3.5    7 
Area = 24.5
Any more rectangle (Y/N)? Y 
Enter length and breadth: 6    4.5
Area = 27
Any more rectangle (Y/N)? N 
```

We have discussed all the three looping statements of C++. Table 7.2 shows a comparison between these statements.

for loop	while loop	do...while loop
Entry controlled loop	Entry controlled loop	Exit controlled loop
Initialisation along with loop definition	Initialisation before loop definition	Initialisation before loop definition
No guarantee to execute the loop body at least once	No guarantee to execute the loop body at least once	Will execute the loop body at least once even though the condition is False

Table 7.2 : Comparison between the looping statements of C++



Let us sum up

The statements providing facilities for taking decisions or for performing repetitive actions in a program are known as control statements. The control statements are the backbones of a computer program. In this chapter we covered the different types of control statements such as selection statements - (if, if...else, if...else if, switch) and iteration statements or loop statements (for, while, do...while). All these control statements will help us in writing complex C++ programs. These statements are essential for various programming operations to be performed.



Learning outcomes

After the completion of this chapter the learner will be able to

- use control statements in C++ for problem solving.
- identify the situation where control statements are used in a program.
- use correct control statements suitable for the situations.
- categorise different types of control statements.
- write C++ programs using control statements.



Lab activity

1. Write a program to input a digit and print it in words.
2. Write a program to print first 'n' odd numbers.
3. Write a program to print the sum of squares of first n odd numbers.
4. Write a program to print every integer between 1 and n which are exactly divisible by m.

Sample questions

Very short answer type

1. Write the significance of break statement in switch statement. What is the effect of absence of break in a switch statement?
2. What will be the output of the following code fragment?

```
for (i=1; i<=10; ++i) ;
cout<<i+5;
```
3. Write a program using for loop that will print the numbers between 1000 and 2000 which are divisible by 132.

- Rewrite the following statement using **while** and **do...while** loops.

```
for (i=1; i<=10; i++) cout<<i;
```
- How many times the following loop will execute?

```
int s=0, i=0;
while(i++<5)
    s+=i;
```

Short answer type

- Consider two program fragments given below.

```
// version 1
cin>>mark;
if (mark >= 90)
    cout<<" A+";
if (mark >= 80 && mark <90)
    cout<<" A";
if (mark >= 70 && mark <80)
    cout<<" B+";
if (mark >= 60 && mark <70)
    cout<<" B";
```

```
//version 2
cin>>mark;
if (mark>=90)
    cout<<" A+";
else if (mark>=80 && mark <90)
    cout<<" A";
else if (mark>=70 && mark <80)
    cout<<" B+";
else if (mark>=60 && mark <70)
    cout<<" B";
```

discuss the advantages of version 2 over version 1.

- Briefly explain the working of a for loop along with its syntax. Give an example of for loop to support your answer.
- Compare and discuss the suitability of three loops in different situations.
- What is wrong with the following while statement if the value of $z = 3$.

```
while(z>=0)
    sum+=z;
```

- Consider the following if else if statement. Rewrite it with switch command.

```
if (a==1)
    cout << "One";
else if (a==0)
    cout << "Zero";
else
    cout << "Not a binary digit";
```

- Write the importance of a loop control variable. Briefly explain the different parts of a loop.

Long answer type

- Explain different types of decision statements in C++.
- Explain different types of iteration statements available in C++ with syntax and examples.