7

Control Statements

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
    - Nesting of loops
- **Jump statements**
    - goto
    - break
    - continue

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 4, 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 another situation. 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 statement 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 mode of execution 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, they must be enclosed in curly braces.

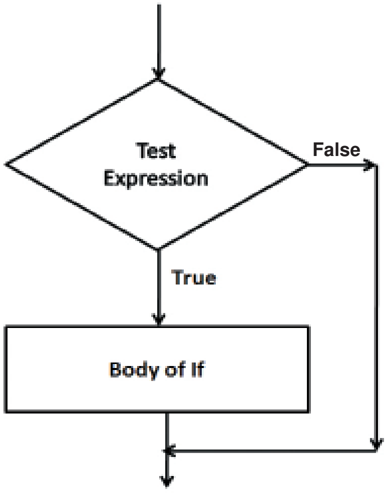


Fig. 7.1 : Working of `if` statement

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 score 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 program.

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 << “You entered a digit\n”;
        cout << “It 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;
}
```

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.

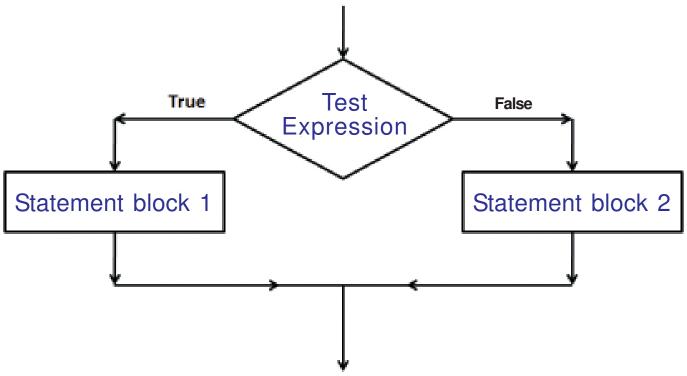


Fig 7.2 : Flowchart of `if - else statement`

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 (i.e. when Test expression returns True)

This statement is executed only when score is less than 18 (i.e. when 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 << "Enter 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 enter a single character for sex and display the gender. If the input is 'M' display "Male" and 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)          outer if
{
    if(age >= 18)          inner if
        cout<<"You are selected for the course!";
}
```

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 score 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 score 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

The `else` 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` is 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 using 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:

Scores	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 your 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";
    return 0;
}
```

The following are the sample outputs of Program 7.5:

Output 1:

```
     Enter your score: 73
     B Grade
```

Output 2:

```
     Enter your score: 25
     E Grade
```

In Program 7.5, initially the test expression `score>=80` is evaluated. Since the input is 73 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;
void main()
{
    int year ;
    cout << "Enter the 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";
    return 0;
}
```

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 the year (in 4-digits): 2000
Leap year
```

Output 2:

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

Output 3:

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

Output 4:

```
Enter the year (in 4-digits): 2004
Leap year
```

Let us 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 display 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 the 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 the 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, 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 integers or character constants. The syntax of **`switch`** statement is as follows:

```
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. We will discuss break statement in detail in Section 7.3.2. 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";
     }
     return 0;
}
```

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` statements 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<<"The given character is a vowel";
                   break;
        default  : cout<<"The 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
    The given character is a vowel
```

Output 2:

```
    Enter the character to check: k
    The 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 have 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. So the following program 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";
        }
```

Score being int type, the expression returns only integer values

Let us have a 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	• Evaluates 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
• `break` statement is required to exit from `switch` 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 `switch`

Table 7.1: Comparison between `switch` *and* `else if` *ladder*

7.1.6 The conditional operator (? :)

As we mentioned in Chapter 6, C++ has a ternary operator. It is the **conditional operator** (**?:**) consisting of the 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;
```

`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 the conditional operator

```
#include <iostream>
using namespace std;
int main()
{
int num1, num2;
cout << "Enter two numbers: ";
cin>> num1 >> num2 ;
(num1>num2)? cout<<num1<<" is larger" : cout<<num2<<" is larger";
return 0;
}
```

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

```
int big = (num1>num2)? num1 : num2;
cout<< big << "is larger";
```

If the test expression evaluates to True, the value of `num1` will be assigned to `big`, otherwise that of `num2`. Here conditional operator is used to construct a conditional expression. The value returned by this expression will be assigned to `big`. The following is a complex form of conditional expression. It gives the largest among three numbers. If `n1`, `n2`, `n3` and `big` are integer variables,

```
big = (n1>n2) ? ( (n1>n3)?n1:n3 ) : ( (n2>n3)?n2:n3);
```

Refer to program 7.4 and see how the above conditional expression replaces the nesting of `if`.

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, and so on).
2. Write a program to perform arithmetic operations using `switch` statement. Accept two operands and a binary arithmetic operator as input.
3. What is the significance of `break` statement within a `switch` statement?
4. Write a program to enter a digit (0 to 9) and display it in words using `switch` statement.
5. Write a program to input a number and check whether it is a multiple of 5 using selection statements and conditional operator.
6. Rewrite the following statement using `if...else` statement

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

7.2 Iteration statements

In Chapter 4, 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 Science. 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 4, we discussed four elements of a loop. Let us refresh them:

1. **Initialisation:** Before entering a loop, its control variable must be initialized. 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.

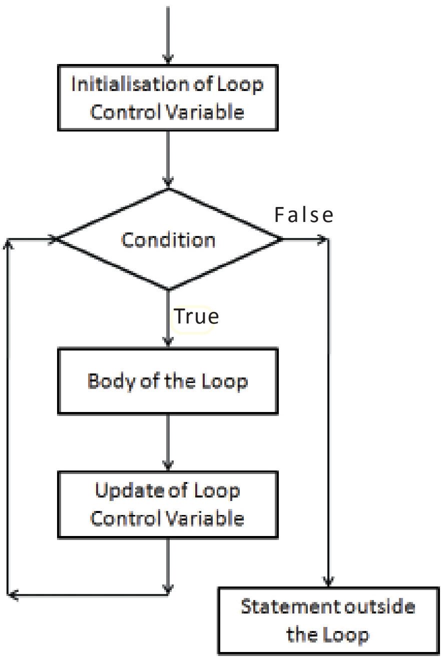
We learnt in Chapter 4 that loops are generally classified into entry-controlled loops and exit-controlled loops. C++ provides three loop statements: **`while`** loop, **`for`** loop and **`do-while`** loop. Let us discuss the working of each one in detail.

7.2.1 while statement

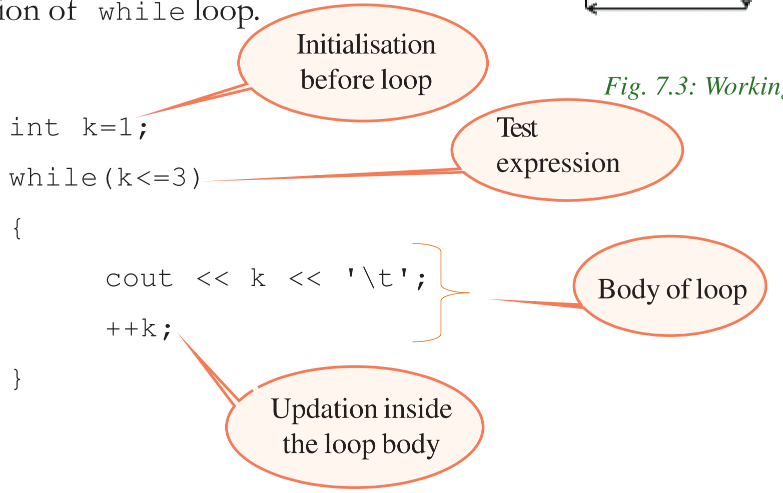
`while` loop is an entry-controlled loop. The condition is checked first and if it is found True the body of the loop will be executed. That is the body will be executed as long as the condition is True. The syntax of **`while`** loop is:

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

Here, `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.


Initialisation of Loop Control Variable
Condition
False
True
Body of the Loop
Update of Loop Control Variable
Statement outside the 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;
}
```

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 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;
    }
    return 0;
}
```

Initialisation of loop variable

Test expression

Body of loop

Updating of loop variable

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<<"\nThe sum of even numbers up to 20 is: "<<sum;
     return 0;
}
```

Loop control variable is updated by adding 2 to the current value

The output of Program 7.12 is given below:

```
The sum of even numbers up to 20 is: 110
```

Let us do

1. *Modify the Program 7.11 to display all odd numbers between 100 and 200.*
2. *Modify the Program 7.12 to find the average 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

`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 suitable in situations where counting is involved. The flowchart given in Figure 7.4 is commonly used to show the execution of `for` statement. At first, the `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.

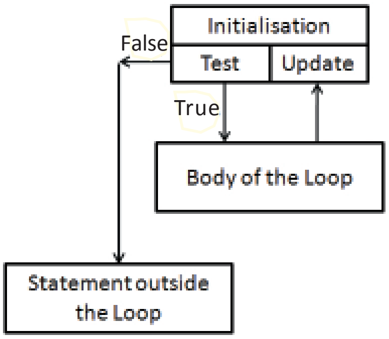


Fig. 7.4: Execution of `for` loop

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 << "  ";
```

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

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: ....

Let us write a program using `for` loop to find the factorial of a number. Factorial of a number, say N, represented as N!, is the product of the first N natural numbers. For example, factorial of 5 (5!) is calculated by 1 × 2 × 3 × 4 × 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;
    long fact=1;
    cout<<"Enter the number: ";
    cin>>n;
    for (i=1; i<=n; ++i)
        fact = fact * i;
    cout << "Factorial of " << n << " is " << fact;
    return 0;
}
```

Initialisation; Test Expression; Updation

Loop body

The following is a sample output of program 7.13

```
Enter the number:  6
Factorial of 6 is 720
```

Another program is given below which gives the class average of scores obtained in Computer Science. Program 7.14 accepts the value for n as the number of students, then reads the scores 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: " << avg;
    return 0;
}
```

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: 47.2
```

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.

While using `for` loops certain points are to be noted. The given 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** and the condition is evaluated to True. Since there is no loop body update takes 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 output statement then 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 above 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.

Another aspect to be noted is that we can provide a number instead of the test expression. If it is zero it will be treated as False, otherwise True.

Check yourself

1. Write a program to find the sum and average of all even numbers between 1 and 49.
2. Write a program to print the numbers between 10 and 50 which are divisible by both 3 and 5.
3. Predict the output of the following code

```
for(int i=1; i<=10; ++i);
       cout << i+2;
```

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);
```

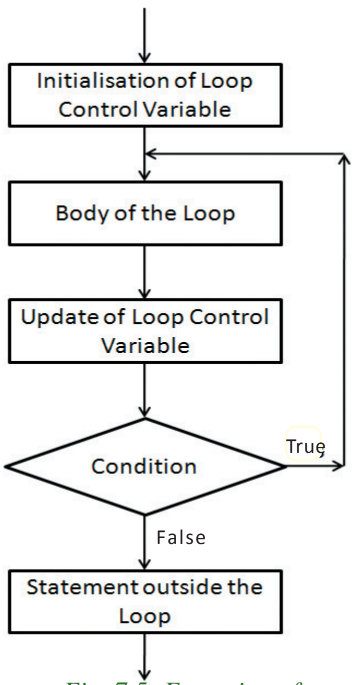
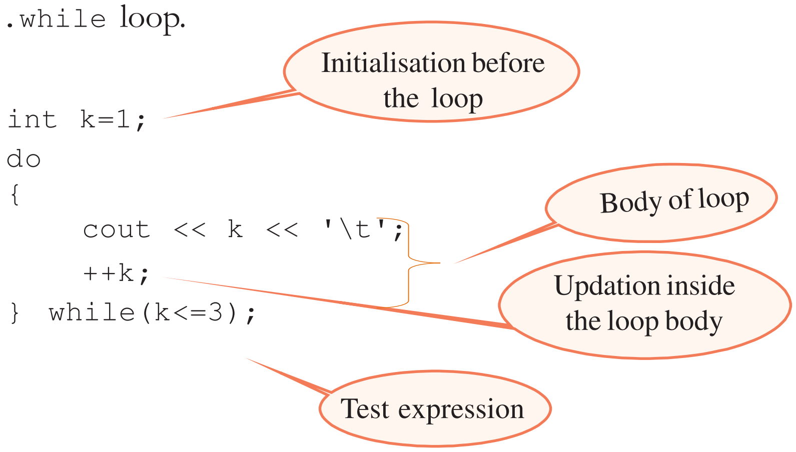

Initialisation of Loop Control Variable
Body of the Loop
Update of Loop Control Variable
Condition
True
False
Statement outside the Loop

Fig. 7.5: Execution of `do..while` loop

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.

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



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

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<=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<=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<=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 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 Program7.15 is given below:

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

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
Initialization along with loop definition	Initialization before loop definition	Initialization 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++

7.2.4 Nesting of loops

Placing a loop inside the body of another loop is called **nesting** of a loop. When we nest two loops, the outer loop counts the number of completed repetitions of the inner loop. Here the loop control variables for the two loops should be different.

Let us observe how a nested loop works. Take the case of a minute-hand and second-hand of a clock. Have you noticed the working of a clock?. While the minute-hand stands still at a position, the second-hand moves to complete one full rotation (say 1 to 60). The minute hand moves to the next position (that is, the next minute) only after the second hand completes one full rotation. Then the second-hand again completes another full rotation corresponding to the minute-hand's current position. For each position of the minute-hand, second-hand completes one full rotation and the process goes on. Here the second hand movement can be treated as the execution of the inner loop and the minute-hand's movement can be treated as the execution of the outer loop.

All types of loops in C++ allow nesting. An example is given to show the working procedure of a nested `for` loop.

```
for( i=1; i<=2; ++i)
{
    for(j=1; j<=3; ++j)
    {
        cout<< "\n" << i << " and " << j;
    }
}
```

Outer loop

Inner loop

Initially value 1 is assigned to the outer loop variable **i**. Its test expression is evaluated to be True and hence the body of the loop is executed. The body contains the inner loop with the control variable **j** and it begins to execute by assigning the initial value 1 to **j**. The inner loop is executed 3 times, for **j** =1, **j**=2, **j**=3. Each time it evaluates the test expression **j<=3** and displays the output since it is True.

```
1 and 1
1 and 2
1 and 3
```

The first 1 is of **i** and the second 1 is of **j**

When the test expression **j<=3** is False, the program control comes out of the inner loop. Now the update statement of the outer loop is executed which makes **i**=2. Then the test expression **i<=2** is evaluated to True and once again the loop body (i.e. the inner loop) is executed. Inner loop is again executed 3 times, for **j**=1, **j**=2, **j**=3 and displays the output.

```
2 and 1
2 and 2
2 and 3
```

After completing the execution of the inner loop, the control again goes back to the update expression of the outer loop. Value of **i** is incremented by 1 (Now **i**=3) and the test expression **i<=2** is now evaluated to be False. Hence the loop terminates its execution. Table 7.3 illustrates the execution of the above given program segment:

Iterations	Outer loop	Inner loop	Output
1	1	1	1 and 1
2	1	2	1 and 2
3	1	3	1 and 3
4	2	1	2 and 1
5	2	2	2 and 2
6	2	3	2 and 3

Table 7.3: Execution of a nested loop

When working with nested loops, the control variable of the outer loop changes its value only after the inner loop is terminated. Let us write a program to display the following triangle using nested loop:

```
*
* *
* * *
* * * *
* * * * *
```

Program 7.16 : To display a triangle of stars

```
#include<iostream>
using namespace std;
int main()
{   int i, j;
    char ch = '*';
    for(i=1; i<=5; ++i)          //outer loop
    {
        cout<< "\n" ;
        for(j=1; j<=i; ++j)      // inner loop
            cout<<ch;
    }
    return 0;
}
```

Let us do

1. *Predict the output of the following program segment:*

```
sum = 0;
for (i=1; i<3; ++i)
{
   for (j=1; j<3; ++j)
       sum = sum +   i * j ;
}
cout<<sum;
```

2. *Write C++ programs to display the following triangles:*

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

7.2.5 Nesting of control statements

We have discussed nesting of `if` and nesting of loops. Any control statement can be nested with another control statement. A loop can contain a selection statement such as `if` or `switch`. Similarly, a selection statement can contain a loop such as `while`, `for` or `do...while`. Program 7.17 contains a loop and its body includes a switch statement. It is the usual style of a menu driven program.

Program 7.17: To input two numbers and perform an arithmetic operation based on user's choice

```
#include<iostream>
using namespace std;
int main()
  {
    char ch;
    float n1, n2;
    cout<<"Enter two numbers: ";
    cin>>n1>>n2;
    do
    {
          cout<<"\nNumber 1: "<<n1<<"\tNumber 2: "<<n2;
          cout<<"\n\t\tOperator Menu";
          cout<<"\n\t1. Addition (+)";
          cout<<"\n\t2. Subtraction (-)";
          cout<<"\n\t3. Multiplication (*)";
          cout<<"\n\t4. Division (/)";
          cout<<"\n\t5. Exit (E)";
          cout<<"\nEnter Option number or operator: ";
          cin>>ch;
          switch(ch)
          {
              case '1' :
              case '+' : cout<<n1<<" + "<<n2<<" = "<<n1+n2;
                         break;
              case '2' :
              case '-' : cout<<n1<<" - "<<n2<<" = "<<n1-n2;
                         break;
              case '3' :
              case '*' : cout<<n1<<" * "<<n2<<" = "<<n1*n2;
                         break;
              case '4' :
              case '/' : cout<<n1<<" / "<<n2<<" = "<<n1/n2;
                         break;
              case '5' :
              case 'E' :
              case 'e' : cout<<"Thank You for using the program";
                         break;
              default  : cout<<"Invalid Choice!!";
          }
    } while (ch!='5' && ch!='E' && ch!='e');
    return 0;
}
```

The following is a sample output of Program 7.17:

```
    Enter two numbers: 25   4
    Number 1: 25          Number 2: 4
                Operator Menu
         1. Addition (+)
         2. Subtraction (-)
         3. Multiplication (*)
         4. Division (/)
         5. Exit (E)
    Enter Option number or operator: 1        User Input
    25 + 4 = 29

    Number 1: 25     Number 2: 4
                 Operator Menu
         1. Addition (+)
         2. Subtraction (-)
         3. Multiplication (*)
         4. Division (/)
         5. Exit (E)
    Enter Option number or operator: /        User Input
    25 / 4 = 6.25

    Number 1: 25     Number 2: 4
                 Operator Menu
         1. Addition (+)
         2. Subtraction (-)
         3. Multiplication (*)
         4. Division (/)
         5. Exit (E)
    Enter Option number or operator: 5        User Input
     Thank You for using the program
```

We will discuss more programs using various combinations of nesting of control statements in the Program gallery section.

7.3 Jump statements

The statements that facilitate the transfer of program control from one place to another are called **jump statements**. C++ provides four types of jump statements that perform unconditional branching in a program. They are **`return`**, **`goto`**, **`break`** and **`continue`** statements. In addition, C++ provides a standard library function **`exit()`** that helps us to terminate a program.

The return statement is used to transfer control back to the calling program or to come out of a function. It will be explained in detail later in Chapter 10. Now, we will discuss the other jump statements.

7.3.1 goto statement

The **`goto`** statement can transfer the program control to anywhere in the function. The target destination of a `goto` statement is marked by a ***label***, which is an identifier.

The syntax of `goto` statement is:

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

where the `label` can appear in the program either before or after `goto` statement. The `label` is followed by a colon (**:**) symbol. For example, consider the following code fragment which prints numbers from 1 to 50.

```
         int  i=1;
start:                     Label
         cout<<i;
         ++i;
         if  (i<=50)
                goto  start;
```

Here, the `cout` statement prints the value 1. After that **`i`** is incremented by 1 (now i=2), then the test expression **i<=50** is evaluated. Since it is True the control is transferred to the statement marked with the label `start`. When the test expression evaluates to False, the process terminates and transfers the program control following the `if` statement.

Let us see another example. Here a number is accepted and tested with a pre-defined value. If it matches, the program continues, otherwise it terminates.

```
  int p;
  cout<<"Enter the Code: ";
  cin>>p;
  if(p!=7755)
     goto end;
  cout<<"Enter the details";
  ........................;
  ........................;          Label
end:
  cout<<"Sorry, the code number is wrong. Try again!";
```

Here the program validates the user input. The program accepts other details only if it is a valid code, otherwise the control goes to the label `end`. It is to be noted that the usage of `goto` is not encouraged in structured programming.

7.3.2 break statement

When a break statement is encountered in a program, it takes the program control outside the immediate enclosing loop (`for`, `while`, `do...while`) or `switch` statement. Execution continues from the statement immediately after the control structure. We have already discussed the impact of `break` in `switch` statement. Let us see how it affects the execution of loops. Consider the following two program segments.

Code segment 1:

```
i=1;
while(i<=10)
{
    cin>>num;
    if (num==0)
        break;
    cout<<"Entered number is: "<<num;
    cout<<"\nInside the loop";
    ++i;
}
cout<<"\nComes out of the loop";
```

The above code fragment allows to input 10 different numbers. During the input if any number happens to be 0, the program control comes out of the loop by skipping the rest of the statements within the loop-body and displays the message "`Comes out of the loop`" on the screen. Let us consider another code segment that uses `break` within a nested loop.

Code segment 2:

```
    for(i=1; i<=5; ++i)    //outer loop
    {
        cout<<"\n";
        for(j=1; j<=i; ++j)      //inner loop
        {
            cout<<"* ";
            if (j==3)
                break;
        }
    }
```

This code segment will display the following pattern:

```
*
*  *
*  *  *
*  *  *
*  *  *
```

Whenever **j** becomes 3, the inner loop terminates

The nested loop executes normally for the value of `i`=1, `i`=2, `i`=3. For each value of `i`, the variable `j` takes values from 1 to `i`. When the value of `i` becomes 4, the inner loop executes for the value of `j` = 1, `j`=2, `j`=3 and comes out from the inner loop on executing the `break` statement.

7.3.3 `continue` statement

`continue` statement is another jump statement used for skipping over a part of the code within the loop-body and forcing the next iteration. The `break` statement forces termination of the loop, but `continue` statement forces next iteration of the loop. The following program segment explains the working of continue statement:

```
    for (i=1; i<=10; ++i)
    {
        if (i==6)
            continue;
        cout<<i<<"\t";
    }
```

This code gives the following output:

```
1     2     3     4     5     7     8     9     10
```

Note that 6 is not in the list. When the value of `i` becomes 6 the `continue` statement is executed. As a result, the output statement is skipped and program control goes to the update expression for next iteration.

A `break` statement inside a loop will abort the loop and transfer control to the statement following the loop. A `continue` statement will just abandon the current iteration and let the loop start next iteration. When `continue` statement is used within `while` and `do...while` loops, care should be taken to avoid infinite execution. Table 7.4 shows a comparison between `break` and `continue` statements.

`break` statement	**`continue` statement**
• Used with `switch` and loops. • Brings the program control outside the `switch` or loop by skipping the rest of the statements within the block. • Program control goes out of the loop even though the test expression is True.	• Used only with loops. • Brings the program control to the beginning of the loop by skipping the rest of the statements within the block. • Program control goes out of the loop only when the test expression becomes False.

Table 7.4: Comparison between the `break` *and* `continue` *statements*

C++ provides a built-in function **`exit()`** which terminates the program itself. The **`exit()`** function can be used in a program only if we include the header file **`cstdlib`** (`process.h`.in Turbo C++). Program 7.18 illustrates the working of this function.

Program 7.18: To check whether the given number is prime or not

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main()
{
    int i, num;
    cout<<"Enter the number: ";
    cin>>num;
    for(i=2; i<=num/2; ++i)
    {
        if(num%i == 0)
        {
            cout<<"Not a Prime Number";
            exit(0);
        }
    }
    cout<<"Prime Number";
    return 0;
}
```

The test expression in the **for** loop of Program 7.18 can be replaced by **i<=sqrt(num)**, where **sqrt()** is a function which gives the square root of the given number. If a number has no factors from 2 to its squrae root, the number will be a prime number. To use **sqrt()**, we have to include the statement **#include<cmath>**

Some sample outputs of program 7.18 are shown below:

Output 1:

```
Enter the number: 17
Prime Number
```

Output 2:

```
Enter the number: 18
Not a Prime Number
```

Check yourself

1. The `goto` statement causes control to go to
   (a) an operator (b) a Label (c) a variable (d) a function
2. A break statement causes an exit
   (a) only from the innermost loop.
   (b) only from the innermost switch.
   (c) from all loops and switches.
   (d) from the innermost loop or switch.
3. The exit( ) function takes the control out of
   (a) the function it appears in.
   (b) the loop it appears in.
   (c) the block it appears in.
   (d) the program it appears in.
4. Name the header file to be included for using exit( ) function.

Program gallery

This section contains a collection of programs which use different control statements for solving various problems. The sample outputs of the programs are also given after each program.

Program 7.19 accepts the three coefficients of a quadratic equation of the form $ax^2 + bx + c = 0$ and calculates its roots. The value of **a** should not be 0 (zero). To solve this problem, the discrminent value of the quadratic equation is to be determined using the formula ($b^2 - 4ac$), to identify the nature of the roots. Formula is also available to find the roots. In this program we use the function `sqrt()` to get the square root of a number. The header file `math.h` is to be included in the program to use this function.

Program 7.19: To find the roots of a quadratic equation

```
#include <iostream>
#include <cmath>// to use sqrt()function
using namespace std;
int main()
{
    float a, b, c, root1, root2, d;
    cout<< "Enter the three coefficients: ";
    cin >> a >> b >> c ;
    if (!a)  // equivalent to if (a == 0)
        cout<<"Value of \'a \' should not be zero\n"
            <<"Aborting!!!!!\n";
    else
    {
        d =b*b-4*a*c;       //beginning of else block
        if (d > 0)
        {
            root1 = (-b + sqrt(d))/(2*a);
            root2 = (-b - sqrt(d))/(2*a);
            cout<<"Roots are REAL and UNEQUAL\n";
            cout<<"Root1 = "<<root1<<"\tRoot2 = "<<root2;
        }
        else if (d == 0)
        {
            root1 = -b/(2*a);
            cout<<"Roots are REAL and EQUAL\n";
            cout<<"Root1 =" <<root1;
        }
        else
            cout<<"Roots are COMPLEX and IMAGINARY";
    }// end of else block of outer if
    return 0;

}
```

Output 1:

```
Enter the three coefficients:    2     3     4
Roots are COMPLEX and IMAGINARY
```

Output 2:

```
Enter the three coefficients:   3     5     1
Roots are REAL and UNEQUAL
Root1 =  -0.232408   Root2 = -1.434259
```

Program 7.20 displays the first N terms of the Fibonacci series. This series begins with terms 0 and 1. The next term onwards will be the sum of the last two terms. The series is 0, 1, 1, 2, 3, 5, 8, 13, ...

Program 7.20: To print n terms of the Fibonacci series

```
#include <iostream>
using namespace std;
int main()
{
    int first=0, second=1, third, n;
    cout<<"\nEnter number of terms in the series: ";
    cin>>n;
    cout<<first<<"\t"<<second;
    for(int i=3; i<=n; ++i)
    {
        third = first + second;
        cout<<"\t"<<third;
        first = second;
        second = third;
    }
    return 0;
}
```

If the variables `first` and `second` are initialised with the values -1 and +1 respectively, we can avoid the `cout` statement for displaying the first two terms.

Output:

```
Enter number of terms in the series: 10
0    1     1     2     3     5     8     13    21    34
```

Program 7.21 reads a number and checks whether it is palindrome or not. A number is said to be palindrome if it is equal to its image. By the term image, we mean a number obtained by reversing the digits of the original number. That is, the image of 163 is 361. Since these two are not equal the number 163 is not palindrome. The number 232 is a palindrome number.

Program 7.21: To check whether the given number is palindrome or not

```
#include <iostream>
using namespace std;
int main()
{
    int num, copy, digit, rev=0;
    cout<<"Enter the number: ";
    cin>>num;
    copy=num;
```

The value of num will be 0 after the completion of loop. That is why the original value is copied into another variable

```
    while(num != 0)
    {
        digit = num % 10;
        rev = (rev * 10)+ digit;
        num = num/10;
    }
    cout<<"The reverse of the number is: "<<rev;
    if (rev == copy)
        cout<<"\nThe given number is a palindrome.";
    else
        cout<<"\nThe given number is not a palindrome.";
    return 0;
}
```

Output 1:

```
Enter the number: 363
The reverse of the number is: 363
The given number is a palindrome.
```

Output 2:

```
Enter the number: 257
The reverse of the number is: 752
The given number is not a palindrome.
```

Program 7.22: To accept n integers and print the largest among them

```
#include <iostream>
using namespace std;
int main()
{
    int num, big, count;
    cout<<"How many Numbers in the list? ";
    cin >> count;
    cout<<"\nEnter first number: ";
    cin >> num;
    big = num;
    for(int i=2; i<=count; i++)
    {
        cout<<"\nEnter next number: ";
        cin >> num;
        if(num > big) big = num;
    }
    cout<<"\nThe largest number is " << big;
    return 0;
}
```

Output:

```
How many Numbers in the list? 5
Enter first number: 23
Enter next number: 12
Enter next number: -18
Enter next number: 35
Enter next number: 18
The largest number is 35
```

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`), iteration statements (`for`, `while`, `do…while`) and also jump statements (`goto`, `break`, `continue`, `exit()` function). All these control statements will help us in writing efficient C++ programs.

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.
- identify different types of jump statements in C++.
- write C++ programs using control statements.

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 the output of the following code fragment be?

```
        for(i=1;i<=10;++i)  ;
        cout<<i+5;
```

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

```
for (i=1; i<=10; i++) cout<<i;
```

4. How many times will the following loop execute?

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

5. Write the name of the header file which contains the exit() function.
6. Which statement in C++ can transfer control of the program to a named label?
7. Write the purpose of `default` statement in `switch` statement.

Short answer type

1. Consider two program fragments given below.

// version 1	//version 2
cin>>mark;	cin>>mark;
if (mark > = 90)	if (mark>=90)
cout<<" A+";	cout<<" A+";
if (mark > = 80 && mark <90)	else if (mark>=80 && mark <90)
cout<<" A";	cout<<" A";
if (mark > = 70 && mark <80)	else if (mark>=70 && mark <80)
cout<<" B+";	cout<<" B+";
if (mark > = 60 && mark <70)	else if (mark>=60 && mark <70)
cout<<" B";	cout<<" B";

Discuss the advantages of version 2 over version 1.

2. Briefly explain the working of a for loop along with its syntax. Give an example of for loop to support your answer.
3. Compare and discuss the suitability of three loops in different situations.
4. Consider the following `if else if` statement. Rewrite it with `switch` statement.

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

5. What is wrong with the following `while` statement if the value of z = 3?

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

6. What will the output of the following code fragments be?

```
for (outer=10; outer > 5; --outer)
{
      for (inner=1; inner<4; ++inner)
            cout<<outer <<"\t"<<inner <<endl;
}
```

7. What will the output of the given code fragments be? Explain.

```
for (n = 1; n <= 10; ++n)
{
      for ( m=1; m <= 5 ; ++m)
            num = n*m;
      cout<<num <<endl;
}
```

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

Long answer type

1. What output will be produced by the following code fragment?

```
int val, res, n=1000;
cin>>val;
res = n+val > 1750 ? 400 : 200;
```

   (a) If the input is 2000

   (b) If the input is 500

2. Write a program to find the sum of digits of a number using
   (a) Entry controlled loop.
   (b) Exit controlled loop.

3. Write a program to print Armstrong numbers less than 1000. (An Armstrong number is a number which is equal to the sum of cubes of its digits. Eg. 153 = $1^3+5^3+3^3$)

4. Explain the different jump statements available in C++.

5. Write a program to produce the following output using nested loop:

```
A
A    B
A    B    C
A    B    C    D
A    B    C    D    E
```

8. Suppose you forgot to write the word `else` in an `if...else` statement. Discuss how it will affect the output of your program?