4

Principles of Programming and Problem Solving

Key concepts

- **Problem solving using computers**
- **Approaches in problem solving**
  - Top down design
  - Bottom up design
- **Phases in programming**
  - Problem identification
  - Algorithms and Flowcharts
  - Coding the program
  - Translation
  - Debugging
  - Execution and testing
  - Documentation
- **Performance evaluation of algorithms**

We have learnt the concept of data processing and the role of computers in data processing. We have also discussed the computer as a system with components such as hardware, software and users. We had a detailed discussion on these components in the previous chapter. Let us recall the definition of software. In its simplest form, we can say that software means a collection of programs to solve problems using computers. As we know, a computer cannot do anything on its own. It must be instructed to perform the desired job. Hence it is necessary to specify a sequence of instructions that the computer must perform to solve a problem. Such a sequence of instructions written in a language that is understood by a computer is called a "computer program". Writing a computer program is a challenging task. However, we can attempt it by procuring the concepts of problem solving techniques and different stages of programming.

4.1 Problem solving using computers

A computer can solve problems only when we give instructions to it. If it understands the tasks contained in the instructions, it will work accordingly. An instruction is an action oriented statement. It tells the computer what operation it should perform. A computer can execute (carry out the task contained in) an instruction only if the task is specified precisely and accurately. As we learned in the previous chapter, there are programmers who develop sequence of instructions for solving problems. Once the program is developed and stored permanently in a computer, we can ask the computer to execute it as and when required.

We should be cautious about the clarity of the logic of the solution and the format of instructions while designing a program, because computer does not possess common sense or intuition. As human beings, we use judgments based on experience, often on subjective and emotional considerations. Such value oriented judgments often depend on what is called "common sense". As opposed to this, a computer exhibits no emotion and has no common sense. That is why we say that computer has no intelligence of its own.

In a way, computer may be viewed as an 'obedient servant'. Being obedient without exercising 'common sense' can be very annoying and unproductive. Take the instance of a master who sent his obedient servant to a post office with the instruction "Go to the post office and buy ten 5 rupees stamps". The servant goes to the post office with the money and does not return even after a long time. The master gets worried and goes in search of him to the post office and finds the servant standing there with the stamps in his hand. When the angry master asks the servant for an explanation, the servant replies that he was ordered to buy ten 5 rupees stamps but not to return with them!

4.2 Approaches in problem solving

A problem may be solved in different ways. Even the approach may be different. In our life, we may seek medical treatment for some diseases. We can consult an allopathic, ayurvedic or homoepathic doctor. Each of their approaches may be different, though all of them are solving the same problem. Similarly in problem solving also different approaches are followed. Let us discuss the two popular designing styles of problem solving – Top down design and Bottom up design.

4.2.1 Top down design

Look at Figure 4.1. If you are asked to draw this picture, how will you proceed? It may be as follows:

1. Draw the outline of the house.
2. Draw the chimney
3. Draw the door
4. Draw the windows

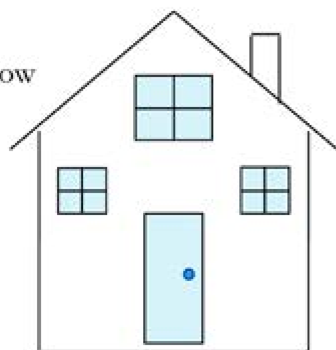


Fig. 4.1 : Lay-out of a House

The procedure described above may be summarised as follows:

While drawing the door in Step 3, the procedure may be as follows:

3.1 Outline of the door

3.2 Shading

3.3 Handle

Similarly the windows may be drawn as follows:

4.1 Outline of the window

4.2 Shading

4.3 Horizontal and Vertical lines

The whole problem (here drawing the picture) is broken down into smaller tasks. Thus four tasks are identified to solve the problem. Some of these tasks (here steps 3 and 4 for drawing the door and windows) are further subdivided. Thus any complex problem can be solved by breaking it down into different tasks and solving each task by performing simpler activities. This concept is known as **top down design** in problem solving.

It is one of the programming approaches that has been proven the most productive. As shown in Figure 4.2, top down design is the process of breaking the overall procedure or task into component parts (modules) and then subdividing each component module until the lowest level of detail is reached. It is also called top down decomposition since we start "at the top" with a general problem and design specific solutions to its sub problems. In order to obtain an effective solution for the main problem, it is desirable that the sub problems (sub programs) should be independent of each other. So, each sub problem can be solved and tested independently.

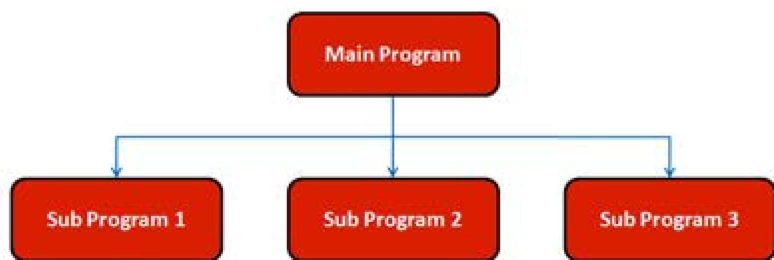


Fig. 4.2 : Decomposition of a problem

The following are the advantages of problem solving by decomposition:

- Breaking the problem into parts helps us to clarify what is to be done in each part.
- At each step of refinement, the new parts become less complicated and therefore, easier to figure out.
- Parts of the solution may turn out to be reusable.
- Breaking the problem into parts allows more than one person to work for the solution.

4.2.2 Bottom up design

Consider the case of constructing a house. We do not follow the top down design, but the bottom up design. The foundation will be the first task and roofing will be the last task. Breaking down of tasks is carried out here too. It is true that some tasks can be carried out only after the completion of some other tasks. However roofing which is the main task will be carried out only after the completion of bottom level tasks.

Similarly in programming, once the overall procedure or task is broken down into component parts (modules) and each component module is further sub divided until the lowest level of detail has been reached, we start solving from the lowest module

onwards. The solution for the main module will be developed only after designing specific solutions to its sub modules. This style of approach is known as **bottom-up design** for problem solving. Here again, it is desirable that the sub problems (subprograms) should be independent of each other.

Let us do

Youth festivals are conducted every year in our schools. Usually the duties and responsibilities are decomposed. Discuss how the tasks are divided and executed to organise the youth festival successfully.

4.3 Phases in programming

As we have seen, problem solving using computer is a challenging task. A systematic approach is essential for this. The programs required can be developed only by going through different stages. Though we have in-born problem solving skills, it can be applied effectively only by proper thinking, planning and developing the logical reasoning to solve the problem. We can achieve this by proceeding through the following stages, in succession:

1. Problem identification
2. Preparing algorithms and flowcharts
3. Coding the program using programming language
4. Translation
5. Debugging
6. Execution and Testing
7. Documentation

Figure 4.3 shows the order of performing the tasks in various stages of programming. Note that the debugging stage is associated with both translation and execution. The activities involved in the seven stages mentioned above are detailed in the following sections.

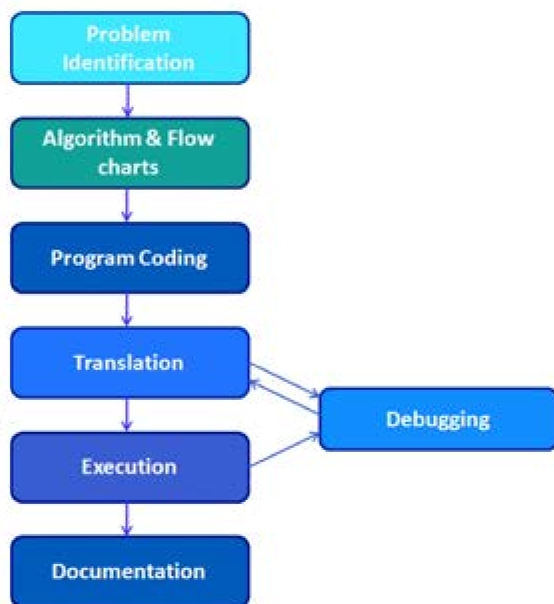


Fig. 4.3 : Phases of Programming

4.3.1 Problem identification

Let us discuss a real life situation; suppose you are suffering from a stomach ache. As you know this problem can be solved by a doctor. Your doctor may ask you some questions regarding the duration of pain, previous occurrence, your diet etc., and examine some parts of your body using the stethoscope, X-ray or scan. All these are a

part of the problem study. After these procedures, your doctor may be able to identify the problem and state it using some medical term. Now the second stage begins with the derivation of some steps for solution known as prescriptions.

It is clear that before deriving the steps for solution, the problem must be analysed. During this phase you will be able to identify the data involved in processing, its type and quantity, formula to be used, activities involved, and the output to be obtained. Once you have studied the problem clearly, and are convinced about the sequence of tasks required for the solution, you can go to the next phase. This is the challenging phase as it exploits the efficiency of the programmer (problem solver).

4.3.2 Algorithms and Flowcharts

Once the problem is identified, it is necessary to develop a precise step-by-step procedure to solve the problem. This procedure is not new or confined to computers. It has been in use for a very long time, and in almost all walks of life. One such procedure taken from real life is described below. It is a cooking recipe of an omlette taken from a magazine.

Ingredients

Eggs - 2 Nos, Onion - 1(small sized, chopped); Green chili - 2 (finely chopped); Oil - 2 tea spoon, Salt - a pinch.

Method

Step 1 : Break the eggs and pour the contents in a vessel and stir.

Step 2 : Mix chopped onion, green chilies and salt with the stirred egg.

Step 3 : Place a pan on the stove and light the stove.

Step 4 : Pour the oil in the pan and wait till it gets heated.

Step 5 : Pour the mixture prepared in step 2 into the pan and wait till the side is fried.

Step 6 : Turn over to get the other side fried well.

Step 7 : Take it out after some seconds.

Result

An omlette is ready to be served with pepper powder.

The recipe given above has the following properties:

1. It begins with a list of ingredients required for making the omlette. These may be called the inputs.
2. A sequence of instructions is given to process the inputs.

3. As a result of carrying out the instructions, some outputs (here, omlette) are obtained.

The instructions given to process the inputs are, however, not precise. They are ambiguous. For example, the interpretation of "till the side is fried" in step 5 and "fried well" in step 6 can vary from person to person. Due to such imprecise instructions, different persons following the same recipe with the same inputs can produce omlettes which differ in size, shape and taste.

The above ambiguities should be avoided while writing steps to solve the problems using the computer.

a. Algorithm

Mathematicians trace the origin of the word algorithm to a famous Arab mathematician, Abu Jafar Mohammed Ibn Musaa Al-Khowarizmi. The word 'algorithm' is derived from the last part of his name Al-Khowarizmi. In computer terminology an **algorithm** may be defined as a finite sequence of instructions to solve a problem. It is a step-by-step procedure to solve a problem, where each step represents a specific task to be carried out. However, in order to qualify an algorithm, a sequence of instructions must possess the following characteristics:

Fig. 4.4 : Abu Jafar Mohammed Ibn Musaa Al-Khowarizmi (780 - 850)

(i) It should begin with instruction(s) to accept inputs. These inputs are processed by subsequent instructions. Sometimes, the data to be processed will be given along with the problem. In such situations, there will be no input instruction at the beginning of the algorithm.

(ii) Use variables to refer the data, where variables are user-defined words consisting of alphabets and numerals that are similar to those used in mathematics. Variables must be used for inputting data and assigning values or results.

(iii) Each and every instruction should be precise and unambiguous. In other words, the instructions must not be vague. It must be possible to carry them out. For example, the instruction "Catch the day" is precise, but cannot be carried out.

(iv) Each instruction must be sufficiently basic such that it can, in principle, be carried out in finite time by a person with paper and pencil.

(v) The total time to carry out all the steps in the algorithm must be finite. As algorithm may contain instructions to repetitively carry out a group of instructions, this requirement implies that the number of repetitions must be finite.

(vi) After performing the instructions given in the algorithm, the desired results (outputs) must be obtained.

To gain insight into algorithms, let us consider a simple example. We have to find the sum and average of any three given numbers. Let us write the procedure for solving this problem. It is given below:

Step 1: Input three numbers.

Step 2: Add these numbers to get the sum

Step 3: Divide the sum by 3 to get the average

Step 4: Print sum and average

Though the procedure is correct, while preparing an algorithm, we have to follow any of the standard formats. Let us see how the procedure listed above can be written in an algorithm style.

Example 4.1: Algorithm to find the sum and average of three numbers

Let A, B, C be variables for the input numbers; and S, Avg be variables for sum and average.

```
Step 1:   Start
Step 2:   Input A, B, C
Step 3:   S = A + B + C
Step 4:   Avg = S / 3
Step 5:   Print S, Avg
Step 6:   Stop
```

The above set of instructions qualifies as an algorithm for the following reasons:

- It has input (The variables A, B and C are used to hold the input data).
- The processing steps are precisely specified (Using proper operators in Steps 3 and 4) and can be carried out by a person using pen and paper.
- Each instruction is basic (Input, Print, Add, Divide) and meaningful.
- It produces two outputs such as sum (S) and average (Avg).
- The beginning and termination points are specified using Start and Stop.

Types of instructions

As we know, a computer can perform only limited types of operations. So we can use only that many instructions to solve problems. Before developing more algorithms, let us identify the types of instructions constituting the algorithm.

- Computer can accept data that we input. So, we can use input instructions. The words **Input**, **Accept** or **Read** may be used for this purpose.
- Computer gives the results as output. So we can use output instructions. The words **Print**, **Display** or **Write** may be used for this purpose.

- Data can directly be stored in a memory location or data may be copied from one location to another. Similarly, results of arithmetic operations on data can also be stored in memory locations. We use assignment (or storing) instruction for this, similar to that used in mathematics. Variables followed by the equal symbol (=) can be used for storing value, where variables refer to memory locations.
- A computer can compare data values (known as logical operation) and make decisions based on the result. Usually, the decision will be in the form of selecting or skipping a set of one or more statements or executing a set of instructions repeatedly.

b. Flowcharts

An idea expressed in picture or diagram is preferred to text form by people. In certain situations, algorithm may be difficult to follow, as it may contain complex tasks and repetitions of steps. Hence, it will be better if it could be expressed in pictorial form. The pictorial representation of an algorithm with specific symbols for instructions and arrows showing the sequence of operations is known as **flowchart**. It is primarily used as an aid in formulating and understanding algorithms. Flowcharts commonly use some basic geometric shapes to denote different types of instructions. The actual instructions are written within these boxes using clear and concise statements. These boxes are connected by solid lines with arrow marks to indicate the flow of operation; that is, the exact sequence in which the instructions are to be executed.

Normally, an algorithm is converted into a flowchart and then the instructions are expressed in some programming language. The main advantage of this two-step approach in program writing is that while drawing a flowchart one is not concerned with the details of the elements of programming language. Hence he/she can fully concentrate on the logic (step-by-step method) of the procedure. Moreover, since a flowchart shows the flow of operations in pictorial form, any error in the logic of the procedure can be detected more easily than in a program. The algorithm and flow chart are always a reference to the programmer. Once these are ready and found correct as far as the logic of the solution is concerned, the programmer can concentrate on coding the operations following the constructs of programming language. This will normally ensure an error-free program.

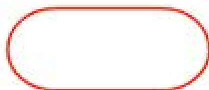
Flowchart symbols

The communication of program logic through flowcharts is made easier through the use of symbols that have standardised meanings. We will only discuss a few symbols that are needed to indicate the necessary operations. These symbols are standardised by the American National Standards Institute (ANSI).

1. Terminal

As the name implies, it is used to indicate the beginning (START) and ending (STOP) in the program logic flow. It is the first symbol and the last symbol in the flowchart.

It has the shape of an ellipse. When it is used as START, an exit flow will be attached. But when it is used as a STOP symbol, an entry flow will be attached.



2. Input / Output

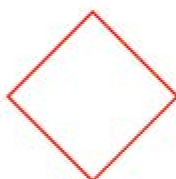
The parallelogram is used as the input/output symbol. It denotes the function of an input/output device in the program. All the input/output instructions are expressed using this symbol. It will be attached with one entry flow and one exit flow.

3. Process

A rectangle is used to represent the processing step. Arithmetic operations such as addition, subtraction, multiplication, division as well as assigning a value to a variable are expressed using this symbol. Assigning a value to a variable means moving data from one memory location to another (e.g. a=b) or moving the result from ALU to memory location (e.g. a=b+5) or even storing a value to a memory location (e.g. a=2). Process symbol also has one entry flow and one exit flow.

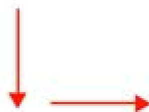
4. Decision

The rhombus is used as decision symbol and is used to indicate a point at which a decision has to be made. A branch to one of two or more alternative points is possible here. All the possible exit paths will be specified, but only one of these will be selected based on the result of the decision. Usually this symbol has one entry flow and two exit flows - one towards the action based on the truth of the condition and the other towards the alternative action.



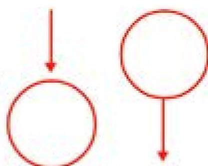
5. Flow lines

Flow lines with arrow heads are used to indicate the flow of operation, that is, the exact sequence in which the instructions are to be executed. The normal flow is from top to bottom and left to right. But in some cases, it can be from right to left and bottom to top. Good practice also suggests that flow lines should not cross each other and such intersections should be avoided wherever possible.



6. Connector

When a flowchart becomes bigger, the flow lines start criss-crossing at many places causing confusion and reducing comprehension of the flowchart. Moreover, when the flowchart becomes too long to fit



into a single page the use of flow lines becomes impossible. Whenever a flowchart becomes complex and the number and direction of flow lines is confusing or it spreads over more than one page, a pair of connector symbols can be used to join the flow lines that are broken. This symbol represents an "entry from", or an "exit to" another part of the flowchart. A connector symbol is represented by a circle and a letter or digit is placed within the circle to indicate the link. A pair of identically labelled connector symbols is commonly used to indicate a continuous flow. So two connectors with identical labels serve the same function as a long flow line. That is, in a pair of identically labelled connectors, one shows an exit to some other chart section and the other indicates an entry from another part of the chart.

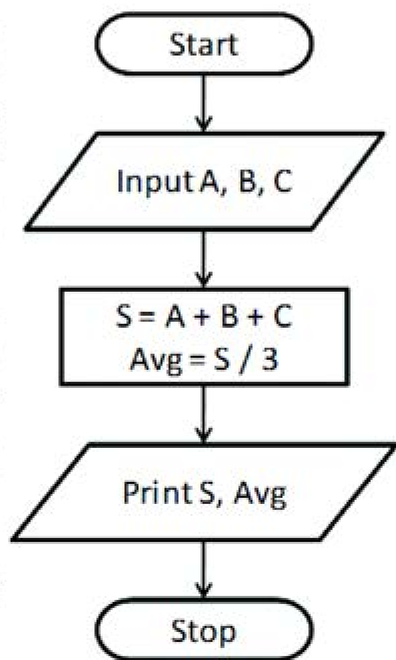


Fig. 4.5 : Flow chart for Sum and Average

Figure 4.5 shows that flowchart of the problem discussed in Example 4.1.

The instruction given in each step in the algorithm is represented using the concerned symbol. Each symbol is labelled properly with the respective instruction. The flow of operations is clearly shown using the flow lines.

Advantages of flowcharts

Flowcharts are beneficial in many ways in program planning.

- **Better communication:** Since a flowchart is a pictorial representation of a program, it is easier for a programmer to explain the logic of the program to some other programmer through a flowchart rather than the program itself.
- **Effective analysis:** The whole program can be analysed effectively through the flowchart as it clearly specifies the flow of the steps constituting the program.
- **Effective synthesis:** If a problem is divided into different modules and the solution for each module is represented in flowcharts separately, they can finally be placed together to visualize the overall system design.
- **Efficient coding:** Once a flowchart is ready, programmers find it very easy to write the concerned program because the flowchart acts as a road map for them. It guides them to go from the starting point of the program to the final point ensuring that no steps are omitted.

Limitations of flowcharts

In spite of their many obvious advantages, flowcharts have some limitations:

- Flowcharts are very time consuming and laborious to draw with proper symbols and spacing, especially for large complex algorithms.
- Owing to the symbol-string nature of flowcharting, any change or modification in the logic of the algorithm usually requires a completely new flowchart.
- There are no standards determining the amount of detail that should be included in a flowchart.

Now let us develop algorithms and draw flowcharts for solving various problems.

Example 4.2: To find the area and perimeter of a rectangle

We know that this problem can be solved, if the length and breadth of the rectangle are given. The result can be obtained by using the following formula:

Perimeter = 2 (Length + Breadth), Area = Length × Breadth

Let L and B be variables for length and breadth; and P, A be variables for perimeter and area.

Step 1: Start
Step 2: Input L, B
Step 3: P = 2 * (L + B)
Step 4: A = L * B
Step 5: Print P, A
Step 6: Stop

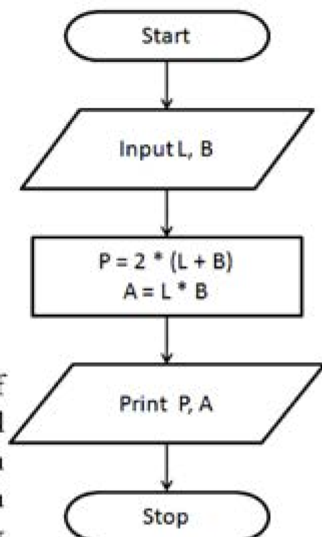


Fig. 4.6 : Flow chart for Area and Perimeter

The flowchart is given in Figure 4.6.

The algorithms developed in Examples 4.1 and 4.2 consist of six instructions each. In both the cases, the instructions will be executed one by one in a sequential fashion as shown in Figure 4.7. The order of execution of instructions is known as flow of control. We can say that the two algorithms follow in a sequential flow of control.

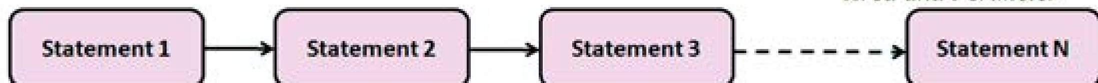


Fig. 4.7 : Sequential flow of control

Let us do

Develop an algorithm and draw the flow chart to input a time in seconds and convert it into the Hr: Min: Sec format. (For example, if 3700 is given as input, the output should be 1 Hr: 1 Min: 40 Sec).

Example 4.3: Find the height of the taller one among two students

Here, two numbers representing the height of two students are to be input. The larger number is to be identified as the result. We know that a comparison between these numbers is to be made for this. The algorithm is given below:

Step 1: Start
Step 2: Input H1, H2
Step 3: If H1 > H2 Then
Step 4: Print H1
Step 5: Else
Step 6: Print H2
Step 7: End of If
Step 8: Stop

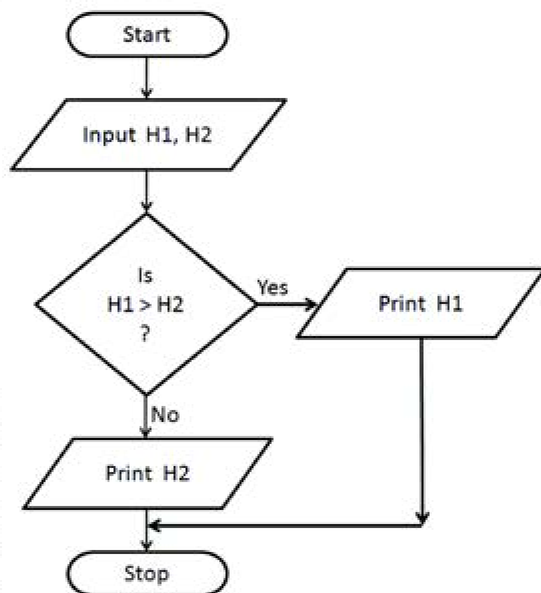


Fig. 4.8 : Flowchart to find larger value

The flowchart of this algorithm is shown in Figure 4.8. This algorithm uses the decision making aspect. In step 3, a condition is checked. Obviously the result may be True or False based on the values of variables H1 and H2. The decision is made on the basis of this result. If the result is True, step 4 will be selected for execution, otherwise step 6 will be executed. Here one of the two statements (either step 4 or step 6) is selected for execution based on the condition. A branching is done in step 3. That is, this algorithm uses the selection structure to solve the problem. As shown in Figure 4.9, the condition branches the flow to one of the two sets based on the result of condition.

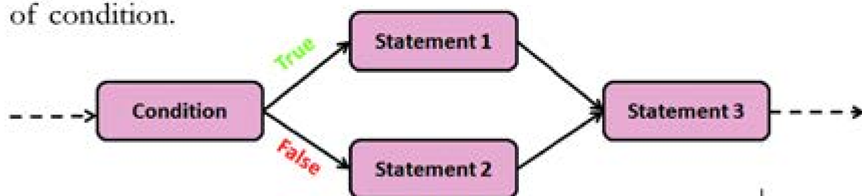


Fig. 4.9 : Selection structure

The working of selection construct is as shown in Figure 4.10. The flow of control comes to the condition; it will be evaluated to True or False. If the condition is True, the instructions given in the true block will be executed and false block will be skipped. But if the condition is False, the true block will be skipped and the false block will be executed. Now let us solve another problem.

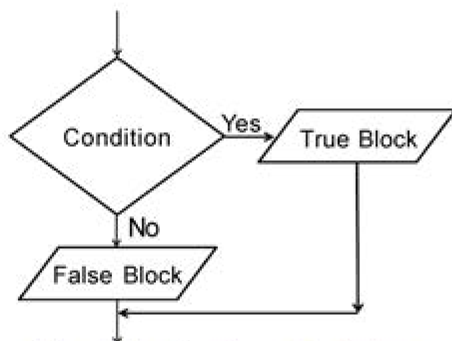


Fig. 4.10 : Flowchart of selection

Example 4.4: To input the scores obtained in 3 unit-tests and find the highest score

Here we have to input three numbers representing the scores and find the largest number among them. The algorithm is given below and flowchart is shown in Figure 4.11.

Step 1: Start
Step 2: Input M1, M2, M3
Step 3: If M1>M2 And M1>M3 Then
Step 4: Print M1
Step 5: Else If M2 > M3 Then
Step 6: Print M2
Step 7: Else
Step 8: Print M3
Step 9: End of If
Step 10: Stop

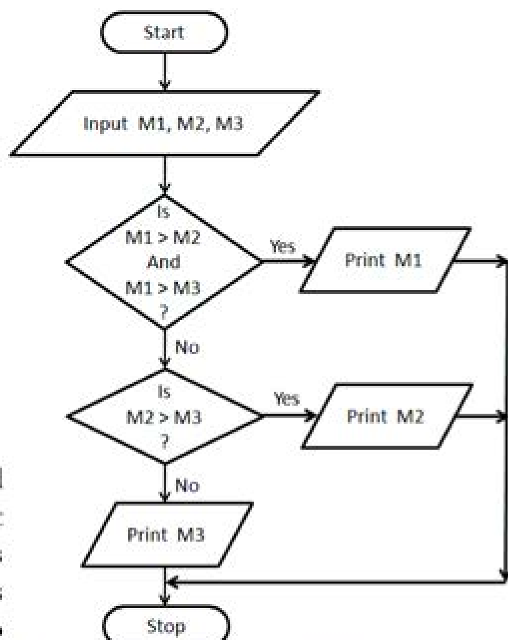


Fig. 4.11 : Flowchart to find the largest of three numbers

This algorithm uses multiple branching based on different conditions. Three different actions are provided, but only one of them is executed. Another point we have to notice is that the first condition consists of two comparisons. This kind of condition is known as compond condition.

Let us do

1. *Develop an algorithm and draw the flowchart to check whether a given number is even or odd.*
2. *Design an algorithm and flow chart to input a day number and display the name of the day. (For example, if 1 is the input, the output should be Sunday. If it is 2, the output should be Monday. If the number is other than 1 to 7, the output should be "Invalid data").*
3. *Based on the evaluation system for standard X, develop an algorithm to accept a score out of 100 and find the grade.*

Now, consider a case in which some task is to be performed in a repeated fashion. Suppose we want to print the first 100 natural numbers. How do we do it? We know that the first number is 1. It should be printed. The next number is obtained by adding 1 to the first number. Again it should be printed. It is clear that the two tasks - printing a number and adding 1 to it - are to be executed repeatedly. The execution should be terminated when the last number is printed. Let us develop the algorithm for this.

Example 4.5: To print the numbers from 1 to 100

Step 1: Start

Step 2: N = 1

Step 3: Print N

Step 4: N = N + 1

Step 5: If N <=100 Then Go To Step 3

Step 7: Stop

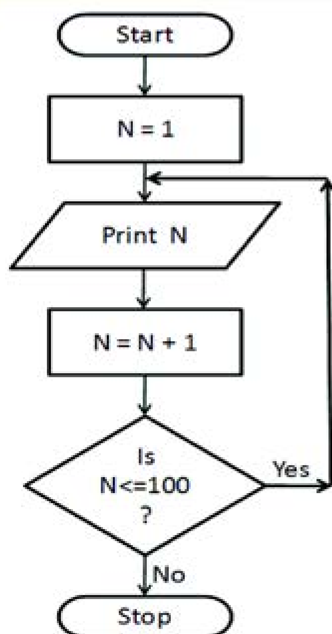


Fig. 4.12 : Flowchart to print numbers from 1 to 100

In the algorithm given as Example 4.5, a condition is checked at step 5. If the condition is found true, the flow of control is transferred back to step 3. So the steps 3, 4 and 5 are executed repeatedly as long as the condition is true. We will say that a loop is formed here. Steps 3, 4 and 5 constitute a loop. The control comes out of the loop only when the condition becomes false. The flowchart of this algorithm is shown in Figure 4.12.

The above algorithm can be simplified as follows:

Step 1: Start

Step 2: N = 1

Step 3: Repeat Steps 4 and 5 While (N<=100)

Step 4: Print N

Step 5: N = N + 1

Step 6: Stop

Note that in step 3, the words "**Repeat**" and "**While**" are used to construct a loop. The statements (step numbers) that need repeated execution is specified with the word "**Repeat**" and the condition is given with "**While**". As the algorithm looks slightly different, the flow chart also differs slightly as in Figure 4.13. The execution style of a loop is shown in Figure 4.14.

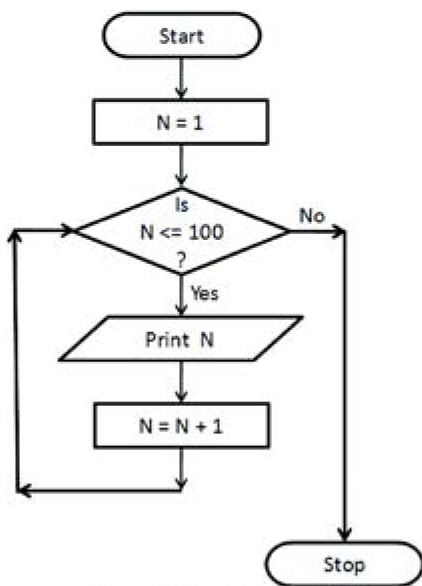


Fig. 4.13 : Flowchart to print numbers from 1 to 100

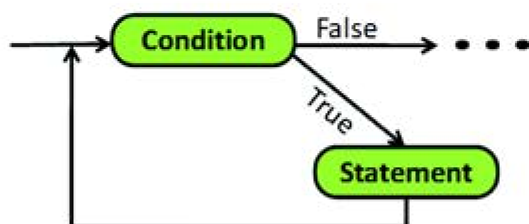


Fig. 4.14 : Looping construct

A loop has four elements. Obviously, one of them is the ***condition***. We know that at least one variable will be used to put up a condition and let us call it loop control variable. Before the condition being checked, the loop control variable should get a value. It is possible through input or assignment. Such an instruction is called **initialisation instruction** for the loop. The third element, called **update instruction**, changes the value of the loop control variable. It is essential; otherwise the execution of the loop will never be terminated. The fourth element is the **loop body**, which is the set of instructions to be executed repeatedly. The flowchart shown in Figure 4.15 depicts the working of looping structure.

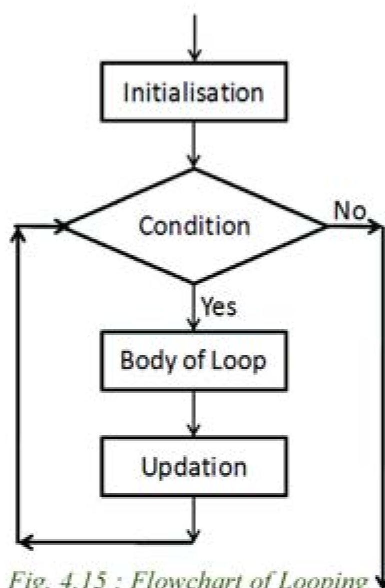


Fig. 4.15 : Flowchart of Looping

The initialisation instruction will be executed first and then the condition will be checked. If the condition is true, the body of the loop will be executed followed by the update instruction. After the execution of the update instruction, the condition will be checked again. This process will be continued until the condition becomes false. The loop that checks the condition before executing the body is called entry-controlled loop. There is another style of looping construct. In this case, the condition will be checked only after the execution of loop-body and update instruction. Such a loop is called exit-controlled loop.

Example 4.6: To print the sum of the first N natural numbers

Here we have to input the value of N. The sum of numbers from 1 to the input number N is to be found out. Let S be the variable to store the sum. Figure 4.16 shows the flowchart of this algorithm

Step 1: Start
Step 2: Input N
Step 3: A = 1, S = 0
Step 4: Repeat Steps 5 and 6 While (A <= N)
Step 5: S = S + A
Step 6: A = A + 1
Step 7: Print S
Step 8: Stop

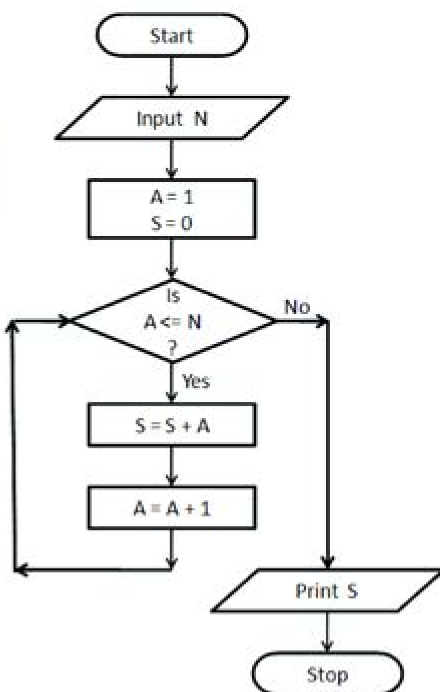


Fig. 4.16 : Flowchart for the sum of first N natural numbers

This algorithm uses an entry-controlled loop. In the next example we can see an algorithm that uses exit-controlled loop for problem solving.

Example 4.7: To print the first 10 multiples of a given number

Step 1: Start

Step 2: Input N

Step 3: A = 1

Step 4: M = A × N

Step 5: Print M

Step 6: A = A + 1

Step 7: Repeat Steps 4 to 6 While (A <= 10)

Step 8: Stop

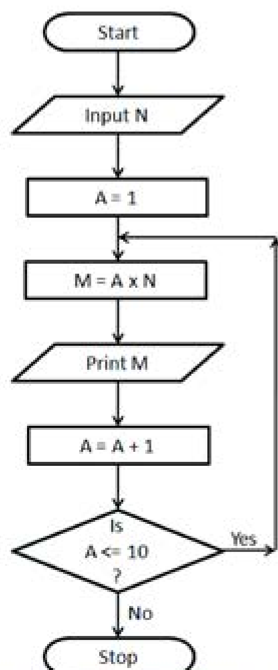


Fig. 4.17 : Flowchart for the first 10 multiples of a number

This algorithm and the corresponding flowchart shown in Figure 4.17 contain a loop in which the condition is checked only after executing the body. Table 4.1 shows comparison between entry controlled loops and exit controlled loops.

Entry Controlled Loop	Exit Controlled Loop
• Condition is checked before the execution of the body	• Condition is checked after the execution of the body
• Body may never be executed.	• Body will surely be executed at least once.
• Suitable when skipping of the body from being executed is required	• Suitable when normal execution of the body is to be ensured.

Table 4.1 : Comparison of Loops

Let us practice with more algorithms and flowcharts for solving the problems that are given as learning activities.

Let us do

Develop an algorithm and draw the flowcharts for the following problems:

1. *To print all even numbers below 100 in the descending order.*
2. *To find the sum of odd numbers between 100 and 200.*
3. *To print the multiplication table of a given number.*
4. *To find the factorial of a number.*
5. *To input a number and check whether it is prime or not.*

4.3.3 Coding the program

Once we have developed the skill to design algorithms and flowcharts, the next step in programming is to express the instructions in a more precise and concise notation. That is, the instructions are to be expressed in a programming language. The process of writing such program instructions to solve a problem is called **coding**. Text editor programs are available to write the code in computer.

A language is a system of communication. We communicate our ideas and emotions to one another through natural languages such as English, Malayalam, etc. Similarly, a computer language is used to communicate between user and computer. A human being who writes a computer program is to be familiar with a language that is understandable to the computer also. We have already seen that computer knows only the binary language which is very difficult for human beings to understand and use. As we saw in Chapter 3, we can use a human friendly language, known as High Level Language (HLL) that looks similar to English. Again there is a facility of using language processors to convert or translate the program written in HLL into machine language. The program written in any HLL is known as **source code**.

Hence, to be a programmer we have to be well-versed in any HLL such as BASIC, COBOL, Pascal, C++ etc. to express the instructions in a program. Each language has its own character set, vocabulary, grammar (we call it syntax) to write programs. Once the program is written using a language, it should be saved in a file (called source file), and then proceed to the next phase of programming.

4.3.4 Translation

While selecting a language for developing source code, certain criteria such as volume of data, complexity in process, usage of files, etc. are to be considered. Once a language is selected and the source code is prepared, it should be translated using the concerned language processor. **Translation** is the process of converting a program written in high level language into its equivalent version in machine language. The compiler or interpreter is used for this purpose. During this step, the syntax errors of the program will be displayed. These errors are to be corrected by opening the file that contains the source code. The source code is again given for compilation (translation). This process will be continued till we get a message such as "No errors or warnings" or "Successful compilation". Now we have a program fully constituted by machine language instructions. This version of the source code is known as **object code** and it will be usually stored in a file by the compiler itself.

Fig. 4.18 : Translation process

Once the object code is obtained it should be present in the system as long as you want the program to be used.

4.3.5 Debugging

Debugging is the stage where programming errors are discovered and corrected. As long as computers are programmed by human beings, the programs will be subject to errors. Programming errors are known as 'bugs' and the process of detecting and correcting these errors is called **debugging**. In general there are two types of errors that occur in a program - syntax errors and logical errors. **Syntax errors** result when the rules or syntax of the programming language are not followed. Such program errors typically involve incorrect punctuation, incorrect word sequence, undefined term, or illegal use of terms or constructs. Almost all language processors detect syntax errors when the program is given for translation. They print error messages that include the line number of the statement having errors and give hints about the nature of the error. In the case of interpreters, the syntax errors will be detected and displayed during execution. The programmer's efficiency in using the language decides the time and effort for the debugging process. The object program will be generated only if all the syntax errors are rectified.

The second type of error, named **logical error**, is due to improper planning of the program's logic. The language processor successfully translates the source code into machine code if there are no syntax errors. During the execution of the program, computer actually follows the program instructions and gives the output as per the instructions. But the output may not be correct. This is known as logical error. When a logical error occurs, all you know is that the computer is not giving the correct output. The computers do not tell us what is wrong. It should be identified by the programmer or user. In order to determine whether or not there is a logical error, the program must be tested. So, let us move on to the next stage of programming.

4.3.6 Execution and testing

As we have seen in the previous section, the program is said to be error-free only when logical errors are also rectified. Hence when the compiled version of the program is formed, it should be executed for testing. The purpose of testing is to determine whether the results are correct. The testing procedure involves running the program to process the test data that will produce 'known results'. That is, the operations involved in the program should be done manually and the output thus obtained should be compared with the one given by the computer. The accuracy of the program logic can be determined by this testing. While selecting the test data, we should ensure that all aspects of the program logic will be tested. Hence the selection of proper test data is important in program testing.

Till now, we have discussed incorrect outputs due to incorrect logic. But there is a chance of another type of error, which will interrupt the program execution. This may be due to the inappropriate data that is encountered in an operation. For example consider an instruction A= B/C. This statement causes interruption in execution if the value of C happens to be zero. In such a situation, the error messages may be displayed by the error-handling function of the language. These errors are known as **Run-time error**. These errors can be rectified by providing instructions for checking the validity of the data before it gets processed by the subsequent instructions in the program.

4.3.7 Documentation

A computerised system cannot be considered to be complete until it is properly documented. In fact documentation is an on-going process that starts in the problem-study phase of the system and continues till its implementation and operation. We can write comments in the source code as part of documentation. It is known as **internal documentation**. It helps the debugging process as well as program modification at a later stage. The logic that we applied in the program may not be remembered when we go through our own program at a later stage. Besides, the program written by one person may need to be modified by some other person in future. If the program is documented, it will help to understand the logic we applied, the reason why a particular statement has been used and so on. However, the documentation part of the program will not be considered by the language processor when you give the program for translation.

Writing comments in programs is only a part of documentation. Another version of documentation is the preparation of system manual and user manual. These are hard copy documents that contain functioning of the system, its requirements etc. and the procedure for installing and using the programs. While developing software for various applications, these manuals are mandatory. This kind of documentation is known as **external documentation**.

Now you have analysed the problem, derived the logic of the solution, expressed in a flow chart, developed the code in a programming language, translated it after removing the syntax errors, checked the accuracy of the output after removing all the possible logical and run-time errors, and we have documented the program.

Check yourself

1. What is an algorithm?
2. Pictorial representation of algorithm is known as ______.
3. Which flow chart symbol is always used in pair?
4. Which flow chart symbol has one entry flow and two exit flows?
5. Program written in HLL is known as ________.
6. What is debugging?
7. What is an object code?

4.4 Performance evaluation of algorithms

We have developed algorithms for solving various problems. You may think that some of these problems would have been solved by following a different logic. Of course, it is true that the same problem can be solved by different sets of instructions. But an efficient programmer is the one who develops algorithms that require minimum computer resources for execution and give results with high accuracy in lesser time. The performance of an algorithm is evaluated based on the concept of time and space complexity. The algorithm which will be executed faster with minimum amount of memory space is considered as the best algorithm for the problem.

Algorithm-1	*Algorithm-2*
Step 1: Start	Step 1: Start
Step 2: Input A, B, C	Step 2: Input A, B, C
Step 3: S = A + B + C	Step 3: S = A + B + C
Step 4: Avg = S / 3	Step 4: Avg = (A + B + C) / 3
Step 5: Print S, Avg	Step 5: Print S, Avg
Step 6: Stop	Step 6: Stop

Table 4.2 : Algorithms to find the sum and average of three numbers

Let us compare the two algorithms given in Table 4.2, developed to find the sum and average of three numbers. The two algorithms differ in step 4. Algorithm-2 uses two steps (steps 3 and 4) for addition operation on the same data. Naturally, that algorithm will take more time for execution than Algorithm-1. So Algorithm-1 is better for coding.

Now let us take another case, where comparison operations are involved for the selection of a statement. We have already discussed an algorithm to find the largest among three numbers in Example 4.4. The two algorithms given in Table 4.3 can also be used for solving the same problem.

Algorithm-1	*Algorithm-2*
Step 1: Start	Step 1: Start
Step 2: Input M1, M2, M3	Step 2 Input M1, M2, M3
Step 3: If M1>M2 And M1>M3 Then	Step 3: If M1 > M2 Then
Step 4: Print M1	Step 4: Big = M1
Step 5: If M2 > M1 And M2 > M3 Then	Step 5: Else
Step 6: Print M2	Step 6: Big = M2
Step 7: If M3 > M1 And M3 > M2 Then	Step 7: If M3 > Big Then Big = M3
Step 8: Print M3	Step 8: Print Big
Step 9: Stop	Step 9: Stop

Table 4.3 : Algorithms to find the largest among three numbers

The algorithm in Example 4.4 has three comparison operations and one logical operation altogether. All these operations are to be carried out only when the largest value is in M3 (the third variable). To identify the speed of execution in each case, let us assume that 1 second is required for one comparison operation. We can see that fastest result will be in 3 seconds and slowest in 4 seconds. So the average speed is 3.5 seconds.

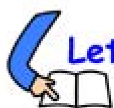
Now let us analyse Algorithm-1 in Table 4.3. There are three **If** statements, each with three comparison operations. If we follow the assumptions specified above, we can see that the result will be obtained in 9 seconds, irrespective of the values in the variables. So the average speed is 9 seconds. But the Algorithm-2 in Table 4.3 uses two If structures. The algorithm shows that whatever be the values in the variables, the time required for comparison will be 2 seconds. Thus the average speed is 2 seconds. So, we can say that the Algorithm-2 is better than the other two.

Let us consider one more case where loop is involved. The two algorithms given in Table 4.4 find the sum of all even numbers and sum of all odd numbers between 100 and 200.

Algorithm-1	*Algorithm-2*
Step 1: Start Step 2: N = 100, ES = 0 Step 3: Repeat Steps 4 to 6 While (N <= 200) Step 4: If Remainder of N/2 = 0 Then Step 5: ES = ES + N Step 6: N = N + 1 Step 7: Print ES Step 8: N = 100, OS = 0 Step 9: Repeat Steps 10 to 12 While (N <= 200) Step 10: If Remainder of N/2 = 1 Then Step 11: OS = OS + N Step 12: N = N + 1 Step 13: Print OS Step 14: Stop	Step 1: Start Step 2: N = 100, ES = 0, OS = 0 Step 3: Repeat Steps 4 to 8 While (N <= 200) Step 4: If Remainder of N/2 = 0 Then Step 5: ES = ES + N Step 6: Else Step 7: OS = OS + N Step 8: N = N + 1 Step 9: Print ES Step 10: Print OS Step 11: Stop

Table 4.4 : Algorithms to find sum of even and odd numbers

Algorithm-1 uses two loops. Obviously, time taken will be double for the intialisation, testing and updation of loop control variable compared to Algorithm-2. From the table, it is clear that Algorithm-2 is better and efficient. So, think divergently and differently to develop logic for solving problems.



Let us sum up

Program is a sequence of instructions written in a computer language. The process of programming proceeds through some stages. Preparation of algorithms and flowcharts help develop the logic. The program written in HLL is known as source code and it is to be converted into machine language. The resultant code is known as object code. The errors occurred in a program has to be removed through a process known as debugging. The translated version is executed by the computer. Proper documentation of the program helps us to modify it at a later stage. While solving problems different logic may be applied, but the performance is measured in terms of time and space complexity.

Learning outcomes

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

- explin various aspects of problem solving.
- develop algorithms for solving problems.
- draw flowcharts to ensure the correctness of algorithms.
- select the best algorithm for solving a problem.

Sample questions

Very short answer type

1. What is an algorithm?
2. What is the role of a computer in problem solving?
3. What is the use of connector in a flow chart?
4. What do you mean by logical errors in a program?

Short answer type

1. What is a computer program? How does an algorithm help to write a program?
2. Write an algorithm to find the sum and average of 3 numbers.
3. Draw a flowchart to display the first 100 natural numbers.
4. What are the limitations of a flow chart?
5. What is debugging?
6. What is the need of documentation for a program?

Long answer type

1. What are the characteristics of an algorithm?
2. What are the advantages of using a flowchart?
3. Briefly explain different phases in programming.

5

Key concepts

- **C++ character set**
- **Tokens**
  - o Keywords
  - o Identifiers
  - o Literals
  - o Punctuators
  - o Operators
- **Integrated Development Environment (IDE)**
  - o Geany IDE

Introduction to C++ Programming

C++ (pronounced "C plus plus") is a powerful, popular object oriented programming (OOP) language developed by Bjarne Stroustrup. The idea of C++ comes from the C increment operator ++, thereby suggesting that C++ is an added (incremented) version of C language.

The C++ language can be used to practice various programming concepts such as sequence, selection and iteration which we have already discussed in Chapter 4. In this chapter, we will have a brief overview of the fundamentals of C++. We will also familiarise different language processor packages that are used to write C++ programs.

Just like any other language, the learning of C++ language begins with the familiarisation of its basic symbols called characters. The learning hierarchy proceeds through words, phrases (expressions), statements, etc. Let us begin with the learning of characters.

5.1 Character set

As we know, the study of any language, such as English, Malayalam or Hindi begins with the alphabet. Similarly, the C++ language also has its own alphabet. With regard to a programming language the alphabet is known as character set. It is a set of valid symbols, called characters that a language can recognize. A character represents

Dr. Bjarne Stroustrup developed C++ at AT&T Bell Laboratories in Murray Hill, New Jersey, USA. Now he is a visiting Professor at Columbia University and holder of the College of Engineering Chair in Computer Science at Texas A&M University. He has received numerous honours. Initial name of this language was 'C with classes'. Later it was renamed to C++, in 1983.

Bjarne Stroustrup

any letter, digit, or any other symbol. The set of valid characters in a language which is the fundamental units of that language, is collectively known as **character set.** The character set of C++ is categorized as follows:

(i)	Letters	: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z a b c d e f g h i j k l m n o p q r s t u v w x y z
(ii)	Digits	: 0 1 2 3 4 5 6 7 8 9
(iii)	Special characters	: + - * / ^ \ ( ) [ ] { } = < > . ' " $, ; : % ! & ? _ (underscore) # @
(iv)	White spaces	: Space bar (Blank space), Horizontal Tab (→), Carriage Return (↵), Newline, Form feed
(v)	Other characters	: C++ can process any of the 256 ASCII characters as data or as literals.

Spaces, tabs and newlines (line breaks) are called white spaces. White space is required to separate adjacent words and numbers.

5.2 Tokens

After learning the alphabet the second stage is learning words constituted by the alphabet (or characters). The term 'token' in the C++ language is similar to the term 'word' in natural languages. **Tokens** are the fundamental building blocks of the program. They are also known as lexical units. C++ has five types of tokens as listed below:

1. Keywords
2. Identifiers
3. Literals
4. Punctuators
5. Operators

5.2.1 Keywords

The words (tokens) that convey a specific meaning to the language compiler are called **keywords**. These are also known as reserved words as they are reserved by the language for special purposes and cannot be redefined for any other purposes. The set of 48 keywords in C++ are listed in Table 5.1. Their meaning will be explained in due course.

asm	continue	float	new	signed	try
auto	default	for	operator	sizeof	typedef
break	delete	friend	private	static	union
case	do	goto	protected	struct	unsigned
catch	double	if	public	switch	virtual
char	else	inline	register	template	void
class	enum	int	return	this	volatile
const	extern	long	short	throw	while

Table 5.1: Keywords of C++

5.2.2 Identifiers

We usually assign names to places, people, objects, etc. in our day to day life, to identify them from one another. In C++ we use identifiers for this purpose. **Identifiers** are the user-defined words that are used to name different program elements such as memory locations, statements, functions, objects, classes etc. The identifiers of memory locations are called ***variables***. The identifiers assigned to statements are called ***labels***. The identifiers used to refer a set of statements are called ***function names***.

While constructing identifiers certain rules are to be strictly followed for their validity in the program. The rules are as follows:

- Identifier is an arbitrary long sequence of letters, digits and underscores (_).
- The first character must be a letter or underscore (_).
- White space and special characters are not allowed.
- Keywords cannot be used as identifiers.
- Upper and lower case letters are treated differently, i.e. C++ is case sensitive.

Examples for some valid identifiers are `Count`, `Sumof2numbers`, `Average_Height`, `_1stRank`, `Main`, `FOR`

The following are some invalid identifiers due to the specified reasons:

`Sum of Digits`	→	Blank space is used
`1styear`	→	Digit is used as the first character
`First.Jan`	→	Special character (`.`) is used
`for`	→	It is a keyword

Let us do

Identify invalid identifiers from the following list and give reasons:
`Data_rec, _data, Idata, data1, my.file, asm, switch, goto, break`

5.2.3 Literals

Consider the case of the Single Window System for the admission of Plus One students. You may have given your date of birth in the application form. As an applicant, your date of birth remains the same throughout your life. Once they are assigned their initial values, they never change their value. In mathematics, we know that the value of π is a constant and the value of gravitational constant 'g' never changes, i.e. it remains $9.8m/s^2$. Like that, in C++, we use the type of tokens called **literals** to represent data items that never change their value during the program run. They are often referred to as constants. Literals can be divided into four types as follows:

1. Integer literals
2. Floating point literals
3. Character literals
4. String literals

Integer literals

Consider the numbers 1776, 707, -273. They are integer constants that identify integer decimal values. The tokens constituted only by digits are called **integer literals** and they are whole numbers without fractional part. The following are the characteristics of integer literals:

- An integer constant must have at least one digit and must not contain any decimal point.
- It may contain either + or – sign as the first character, which indicates whether the number is positive or negative.
- A number with no sign is treated as positive.
- No other characters are allowed.

Classify the following into valid and invalid integer constants and give reasons for the ivalidity:

77,000	70	314.	-5432	+15346
+23267	-.7563	-02281+0	1234E56	-9999

In addition to decimal numbers (base 10), C++ allows the use of octal numbers (base 8) and hexadecimal numbers (base 16) as literals (constants). To express an octal number we have to precede it with a 0 (zero character) and in order to express a hexadecimal number we have to precede it with the characters 0x (zero, x). For example, the integer constants 75, 0113 and 0x4B are all equivalent to each other. All of these represent the same number 75 (seventy-five), expressed as a base-10 numeral, octal numeral and hexadecimal numeral, respectively.

Floating point literals

You may have come across numbers like 3.14159, 3.0×10^{8}, 1.6×10^{-19} and 3.0 during your course of study. These are four valid numbers. The first number is π (Pi), the second one is the speed of light in meter/sec, the third is the electric charge of an electron (an extremely small number) – all of them are approximated, and the last one is the number three expressed as a floating-point numeric literal.

Floating point literals, also known as real constants are numbers having fractional parts. These can be written in one of the two forms called fractional form or exponential form.

A real constant in fractional form consists of signed or unsigned digits including a decimal point between digits. The rules for writing a real constant in fractional form are given below:

- A real constant in fractional form must have at least one digit and a decimal point.
- It may also have either + (plus) or – (minus) sign preceding it.
- A real constant with no sign is assumed to be positive.

A real constant in exponential form consists of two parts: *mantissa* and *exponent*. For instance, 5.8 can be written as 0.58×10^{1} = 0.58E1 where mantissa part is 0.58 (the part appearing before **E**) and exponential part is 1 (the part appearing after **E**). The number E1 represents 10^{1}. The rules for writing a real constant in exponential form are given below:

- A real constant in exponent form has two parts: a mantissa and an exponent.
- The mantissa must be either an integer or a valid fractional form.

- The mantissa is followed by a letter **E** or **e** and the exponent.
- The exponent must be an integer.

The following are valid real constants.

52.0	107.5	-713.8	-.00925
453.E-5	1.25E08	.212E04	562.0E09
152E+8	1520E04	-0.573E-7	-.097

Some invalid real constants are given along with the reason:

58,250.262 (Comma is used), 5.8E (No exponent part), 0.58E2.3 (Fractional number is used as exponent).

Let us do

Classify the following into valid and invalid real constants and justify your answer:

77, 00,000	7.0	3.14	-5.0E5.4	+53.45E-6
+532.67.	.756E-3	-0.528E10	1234.56789	34,56.24
4353	+34/2	5.6E	4356	0

Character literals

When we want to store the letter code for gender usually we use 'f' or 'F' for *Female* and 'm' or 'M' for *Male*. Similarly, we may use the letter 'y' or 'Y' to indicate *Yes* and the letter 'n' or 'N' to indicate *No*. These are single characters. When we refer a single character enclosed in single quotes that never changes its value during the program run, we call it a **character literal** or **character constant**.

Note that x without single quote is an identifier whereas `'x'` is a character constant. The value of a single character constant is the ASCII value of the character. For instance, the value of `'c'` will be 99 which is the ASCII value of `'c'` and the value of `'A'` will be the ASCII value 65.

C++ language has certain non-graphic character constants, which cannot be typed directly from the keyboard. For example, there is no way to express the Carriage Return or Enter key, Tab key and Backspace key. These non-graphic symbols can be represented by using **escape sequences**, which consists of a backslash (\) followed by one or more characters. It should be noted that even though escape sequences contain more than one character enclosed in single quotes, it uses only one corresponding ASCII code to represent it. That is why they are treated as character constants. Table 5.2 lists escape sequences and corresponding characters.

In Table 5.2, we can also see sequences representing `\'`, `\"` and `\?`. These characters can be typed from the keyboard but when used without escape sequence, they carry a special meaning and have a special purpose. However, if these are to be displayed or printed as it is, then escape sequences should be used. Examples of some valid character constants are: `'s'`, `'S'`, `'$'`, `'\n'`, `'+'`, `'9'`

Some invalid character constants are also given with the reason for invalidity:

A (No single quotes), '82' (More than one character), "K" (Double quotes instead of single quotes), '\g' (Invalid escape sequence or Multiple characters).

Escape Sequence	Corresponding Non-graphic character
`\a`	Audible bell (alert)
`\b`	Back Space
`\f`	Form feed
`\n`	New line or Line feed
`\r`	Carriage Return
`\t`	Horizontal Tab
`\v`	Vertical Tab
`\\`	Back slash
`\'`	Single quote
`\"`	Double quote
`\?`	Question mark
`\0`	Null character

Table 5.2 : Escape Sequences

C++ represents Octal Number and Hexadecimal Number with the help of escape sequences. The \0n and \xHn represent a number in the Octal Number System and the Hexadecimal Number System respectively.

String literals

Nandana is a student and she lives in Bapuji Nagar. Here, "Nandana" is the name of a girl and "Bapuji Nagar" is the name of a place. These kinds of data may need to be processed with the help of programs. Such data are considered as string constants and they are enclosed within double quotes. A sequence of one or more characters enclosed within a pair of double quotes is called **string constant**. For instance, `"Hello friends"`, `"123"`, `"C++"`, `"Baby\'s Day Out"`, etc. are valid string constants.

Let us do

Classify the following into different categories of literals.

```
'a'              "rita"   -124      12.5   -12e-1
"raju\'s pen"    0        -11.999   '\\'   32760
```

5.2.4 Punctuators

In languages like English, Malayalam, etc. punctuation marks are used for grammatical perfection of sentences. Consider the statement: *Who developed C++?* Here '?' is the punctuation mark that tells that the statement is a question. Similarly at the end of each sentence we put a full stop (.). In the same way C++ also has some special symbols that have syntactic or semantic meaning to the compiler. These are called **punctuators**. Examples are: `# ; ' " ( ) [ ] { }`. The purpose of each punctuator will be discussed later.

5.2.5 Operators

When we have to add 5 and 3, we express it as 5 + 3. Here + is an operator that represents the addition operation. Similarly, C++ has a rich collection of operators. An **operator** is a symbol that tells the compiler about a specific operation. They are the tokens that trigger some kind of operation. The operator is applied on a set of data called **operands**. C++ provides different types of operators like arithmetic, relational, logical, assignment, conditional, etc. We will discuss more about operators in the next chapter.

Let us do

Classify the following into different categories of tokens.

```
/       -124               +     -12e-1    '"KL01"
Sum     "raju\'s pen"      if    rita      '\\'
break    }
```

5.3 Integrated Development Environment (IDE)

Now we have learned the basic elements of a C++ program. Before we start writing C++ programs, we must know where we will type this program. Like other programming languages, a text editor is used to create a C++ program. The compilers such as GCC (GNU Compiler Collection), Turbo C++, Borland C++, and many other similar compilers provide an Integrated Development Environment (IDE) for developing C++ programs. Many of these IDEs provide facilities for typing, editing, searching, compiling, linking and executing a C++ program. We use Geany IDE (IT@School Ubuntu Linux 12.04) for the purpose of illustrating the procedure for coding, compiling and executing C++ programs.

GCC with Geany IDE

GCC compiler is a free software available with Linux operating system. GCC stands for GNU Compiler Collection and is one of the popular C++ compilers which

works with ISO C++ standard. Geany is a cross-platform IDE for writing, compiling and executing C++ programs.

A. Opening the edit window

The edit window of Geany IDE can be opened from the **Applications** menu of Ubuntu Linux by proceeding as follows:

Applications → Programming → Geany

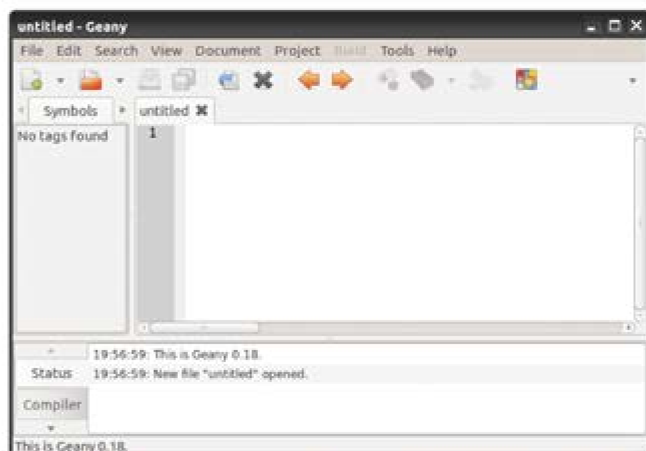
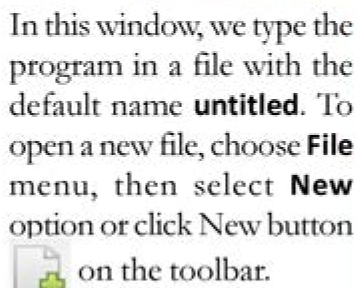


Fig. 5.1: Opening screen of Geany IDE in Ubuntu LInux

Geany IDE opens its window as shown in Figure 5.1. It has a title bar, menu bar, toolbar, and a code edit area. We can see a tab named **untitled**, which indicates the file name for the opened edit area. If we use Geany 1.24 on Windows operating system, the opening window will be as shown in Figure 5.2. We can see that both of these are quite the same.

In this window, we type the program in a file with the default name **untitled**. To open a new file, choose **File** menu, then select **New** option or click New button



on the toolbar.

b. Saving the program

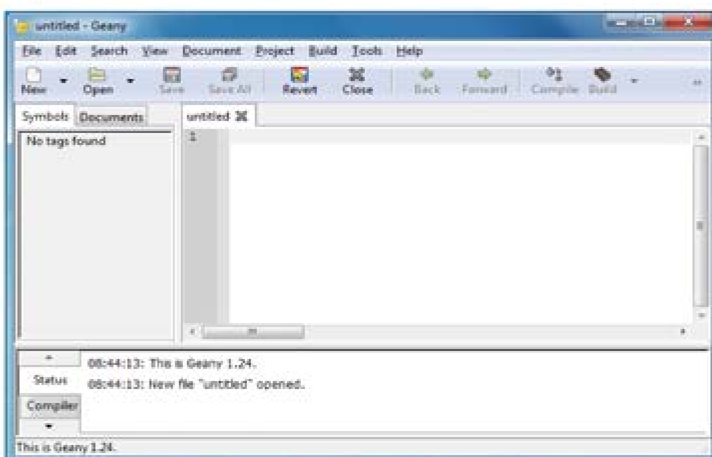


Fig. 5.2: Opening screen of Geany IDE 1.24 in Windows OS

Once a file is opened, enter the C++ program and save it with a suitable file name with extension **.cpp**. GCC being a collection of compilers, the extension decides which compiler is to be selected for the compilation of the code. Therefore we have to specify the extension without fail. If we give the file name before typing the program, GCC provides different colours automatically to distinguish the types of tokens used in the program. It also uses indentation to identify the level of statements in the source code. We will disucss the concept of indentation later.

Let us write a simple program given as Program 5.1 and save with the name **welcome.cpp**.

Program 5.1: A program to familiarise the IDE

```
// my first C++ program
#include<iostream>
using namespace std;
int main()
{
      cout << "Welcome to the world of C++";
      return 0;
}   //end of program
```

The IDE window after entering Program 5.1 is shown in Figure 5.3. Observe the diffrence in colours used for the tokens.

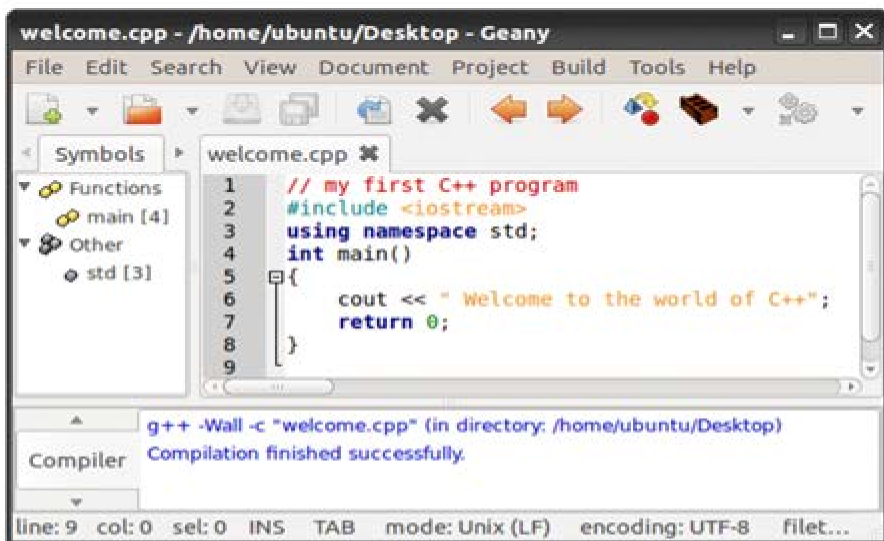
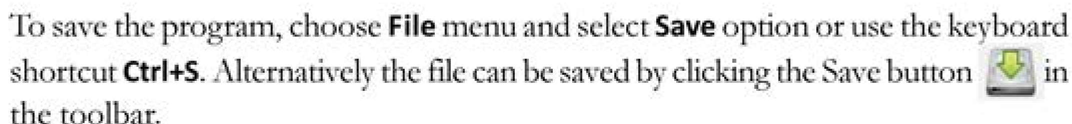


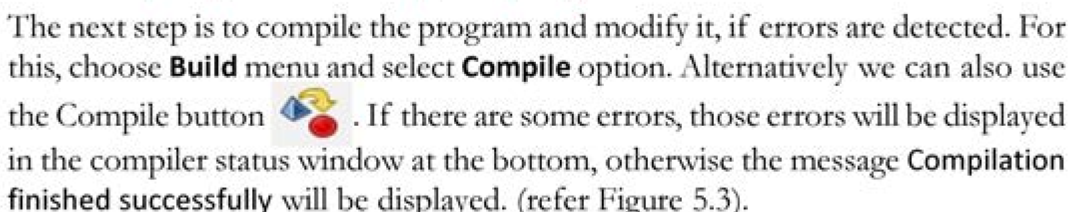
Fig. 5.3: Program saved with a name in Geany IDE

To save the program, choose **File** menu and select **Save** option or use the keyboard shortcut **Ctrl+S**. Alternatively the file can be saved by clicking the Save button  in the toolbar.

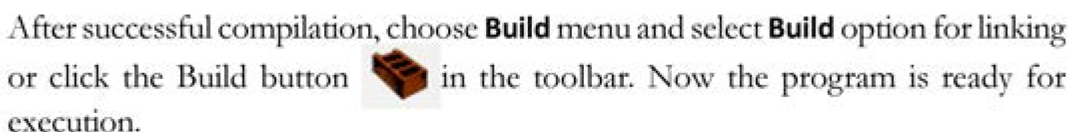
It is a good practice to save the program every now and then, just by pressing **Ctrl+S**. This helps to avoid the loss of data due to power failures or due to unexpected system errors. Once the program typing is completed, it is better to save the file before compiling or modifying. Copying the files from the temporary volatile primary memory to permanent non volatile secondary memory for storage is known as saving the program.

C++ program files should have a proper extension depending upon the implementation of C++. Different extensions are followed by different compilers. Examples are .cpp, .cxx, .cc, .c++

C. Compiling and linking the program

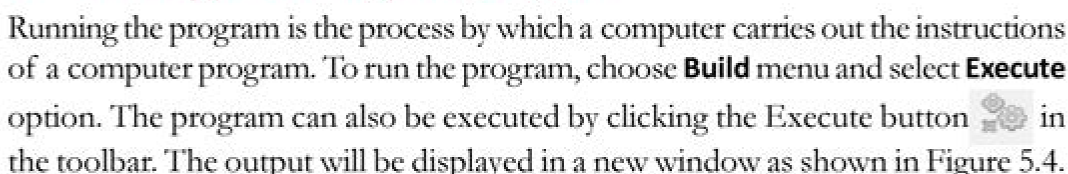


The next step is to compile the program and modify it, if errors are detected. For this, choose **Build** menu and select **Compile** option. Alternatively we can also use the Compile button . If there are some errors, those errors will be displayed in the compiler status window at the bottom, otherwise the message **Compilation finished successfully** will be displayed. (refer Figure 5.3).



After successful compilation, choose **Build** menu and select **Build** option for linking or click the Build button in the toolbar. Now the program is ready for execution.

D. Running/Executing the program



Running the program is the process by which a computer carries out the instructions of a computer program. To run the program, choose **Build** menu and select **Execute** option. The program can also be executed by clicking the Execute button in the toolbar. The output will be displayed in a new window as shown in Figure 5.4.

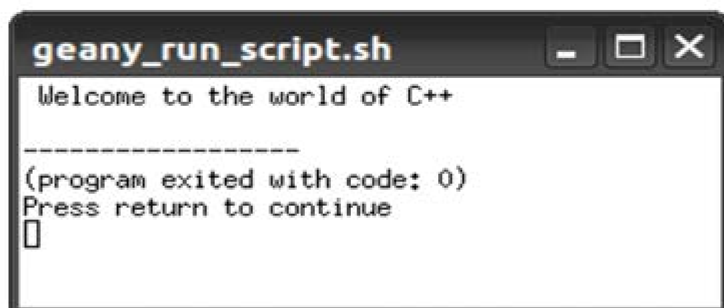
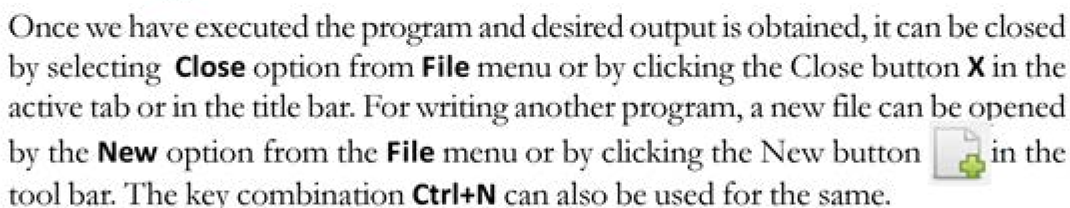


Fig. 5.4: Output window

E. Closing the IDE



Once we have executed the program and desired output is obtained, it can be closed by selecting **Close** option from **File** menu or by clicking the Close button **X** in the active tab or in the title bar. For writing another program, a new file can be opened by the **New** option from the **File** menu or by clicking the New button in the tool bar. The key combination **Ctrl+N** can also be used for the same.

After developing program, we can come out of Geany IDE by choosing **File** menu and selecting **Quit** option. The same can be achieved by clicking the Close button of the IDE window or by using the key combination **Ctrl+Q**.

1. *Write a program to print the message "SMOKING IS INJURIOUS TO HEALTH" on screen.*
2. *Write a program to display the message "TOBACCO CAUSES CANCER" on monitor.*

Let us sum up

C++ was developed by Bjarne Stroustrup in early 1980s. C++ has its own character set. Tokens are the smallest unit of a program and are constituted by one or more characters in C++. There are five types of tokens namely keywords, identifiers, literals, punctuators and operators. Programs are written in computer with the help of an editor. Software like GCC and Geany IDE provide facilities to enter the source code in the computer, compile it and execute the object code.

Learning outcomes

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

- list the C++ character set.
- categorise various tokens.
- identify keywords.
- write valid identifiers.
- classify various literals.
- identify the main components of Geany IDE.
- write, compile and run a simple program.

Sample questions

Very short answer type

1. What are the different types of characters in C++ character set?
2. What is meant by escape sequences?
3. Who developed C++?
4. What is meant by tokens? Name the tokens available in C++.
5. What is a character constant in C++ ?
6. How are non-graphic characters represented in C++? Give an example.
7. Why are the characters \ (slash), ' (single quote), " (double quote) and ? (question mark) typed using escape sequences?
8. Which escape sequences represent newline character and null character?
9. An escape sequence represents ______ characters.
10. Which of the following are valid character/string constants in C++?
    `'c'` `'anu'` `"anu"` `mine` `'main's'` `" "`
    `'char` `'\ \'`
11. What is a floating point constant? What are the different ways to represent a floating point constant?
12. What are string-literals in C++? What is the difference between character constants and string literals?
13. What is the extension of C++ program file used for running?
14. Find out the invalid identifiers among the following. Give reason for their invalidity
    a) Principal amount b) Continue c) Area d) Date-of-join e) 9B
15. A label in C++ is ________.
    a) Keyword b) Identifier c) Operator d) Function
16. The following tokens are taken from a C++ program. Fill up the given table by placing them at the proper places
    (`int`, `cin`, %, `do` , = , `"break"` , `25.7`, `digit`)

Keywords	Identifiers	Literals	Operators

Short answer type

1. Write down the rules governing identifiers.
2. What are tokens in C++ ? How many types of tokens are allowed in C++?List them.
3. Distinguish between identifiers and keywords.
4. How are integer constants represented in C++? Explain with examples.
5. What are character constants in C++? How are they implemented?

Long answer type

1. Briefly describe different types of tokens.
2. Explain different types of literals with examples.
3. Briefly describe the Geany IDE and its important features.

6

Data Types and Operators

Key concepts

- Concept of data types
- C++ data types
- Fundamental data types
- Type modifiers
- Variables
- Operators
  - Arithmetic
  - Relational
  - Logical
  - Input/Output
  - Assignment
  - Arithmetic assignment
  - Increment and decrement
  - Conditional
  - sizeof
  - Precedence of operators
- Expressions
  - Arithmetic
  - Relational
  - Logical
- Type conversion
- Statements
  - Declaration
  - Assignment
  - Input /Output
- Structure of a C++ program
  - Pre-processor directives
  - Header files
  - Concept of namespace
  - The main() function
  - A sample program
- Guidelines for coding

In the previous chapter we familiarised ourselves with the IDE used for the development of C++ programs and also learnt the basic building blocks of C++ language. As we know, data processing is the main activity carried out in computers. All programming languages give importance to data handling. The input data is arranged and stored in computers using some structures. C++ has a predefined template for storing data. The stored data is further processed using operators. C++ also makes provisions for users to define new data types, called user-defined data types.

In this chapter, we will explore the main concepts of the C++ language like data types, operators, expressions and statements in detail.

6.1 Concept of data types

Consider the case of preparing the progress card of a student after an examination. We need data like admission number, roll number, name, address, scores in different subjects, the grades obtained in each subject, etc. Further, we need to display the percentage of marks scored by the student and the attendance in percentage. If we consider a case of scientific data processing, it may require data in the form of numbers representing the velocity of light (3×10^8 m/s), acceleration due to gravity (9.8 m/s), electric charge of an electron (-1.6×10^{-19}) etc.

From these cases, we can infer that data can be of different types like character, integer number, real number, string, etc. In the last chapter we saw that any valid character of C++ enclosed in single quotes represents character data in C++. Numbers without fractions represent integer data. Numbers with fractions represent floating point data and anything enclosed in double quotes represents a string data. Since the data to be dealt with are of many types, a programming language must provide ways and facilities to handle all types of data. C++ provides facilities to handle different types of data by providing data type names. **Data types** are the means to identify the nature of the data and the set of operations that can be performed on the data. Various data types are defined in C++ to differentiate these data characteristics.

In Chapter 4, we used variables to refer data in algorithms. Variables are also used in programs for referencing data. When we write programs in the C++ language, variables are to be declared before their use. Data types are necessary to declare these variables.

6.2 C++ data types

C++ provides a rich set of data types. Based on nature, size and associated operations, they are classified as shown in Figure 6.1. Basically, they are classified into fundamental or built-in data types, derived data types and user-defined data types.

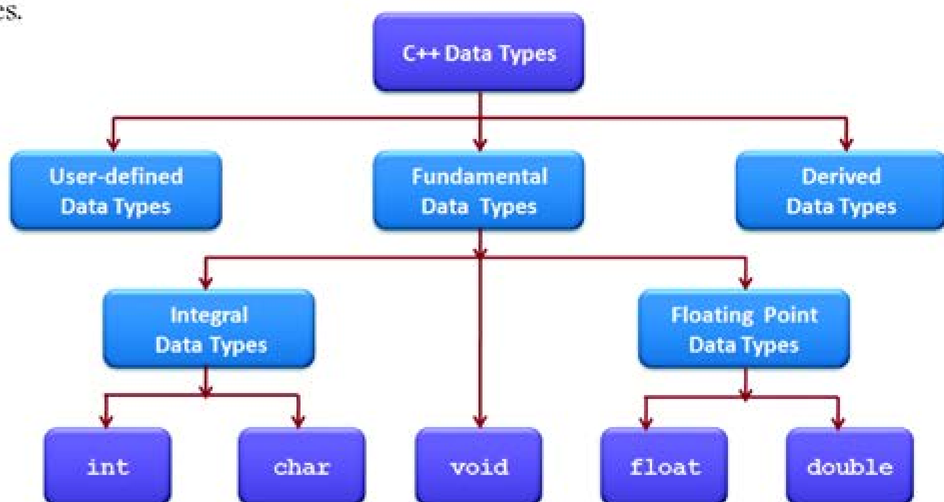


Fig 6.1 : Classification of C++ data types

Fundamental data types

Fundamental data types are defined in C++ compiler. They are also known as built-in data types. They are atomic in nature and cannot be further decomposed of. The five fundamental data types in C++ are `char`, `int`, `float`, `double` and `void`. Among these, `int` and `char` comes under integral data type as they can handle

only integers. The numbers with fractions (real numbers) are generally known as floating type and are further divided into `float` and `double` based on precision and range.

User-defined data types

C++ is flexible enough to allow programmers to define their own data types. Structure (`struct`), enumeration (`enum`), union, class, etc. are examples for such data types.

Derived data types

Derived data types are constructed from fundamental data types through some grouping or alteration in the size. Arrays, pointers, functions, etc. are examples of derived data types.

6.3 Fundamental data types

Fundamental data types are basic in nature. They cannot be further broken into small units. Since these are defined in compiler, the size (memory space allocated) depends on the compiler. We use the compiler available in GCC and hence the size as well as the range of data supported by the data type are given accordingly. It may be different if you use other compilers like Turbo C++ IDE. The five fundamental data types are described below:

`int` data type (*for integer numbers*)

Integers are whole numbers without a fractional part. They can be positive, zero or negative. The keyword **`int`** represents integer numbers within a specific range. GCC allows 4 bytes of memory for integers belonging to `int` data type. So the range of values that can be represented by `int` data type is from -2147483648 to +2147483647. The data items 6900100, 0, -112, 17, -32768, +32767, etc. are examples of `int` data type. The numbers 2200000000 and -2147483649 do not belong to `int` data type as they are out of the allowed range.

`char` data type (*for character constants*)

Characters are the symbols covered by the character set of the C++ language. All letters, digits, special symbols, punctuations, etc. come under this category. When these characters are used as data they are considered as `char` type data in C++. We can say that the keyword **`char`** represents character literals of C++. Each `char` type data is allowed one byte of memory. The data items 'A', '+', '\t','0', etc. belong to `char` data type. The `char` data type is internally treated as integers, because computer recognises the character through its ASCII code. Character data is stored in the memory with the corresponding ASCII code. As ASCII codes are integers and need to be stored in one byte (8 bits), the range of `char` data type is from -128 to +127.

`float` data type (*for floating point numbers*)

Numbers with a fractional part are called floating point numbers. Internally, floating-point numbers are stored in a manner similar to scientific notation. The number 47281.97 is expressed as 0.4728197×10^5 in scientific notation. The first part of the number, 0.4728197 is called the mantissa. The power 5 of 10 is called exponent. Computers typically use exponent form (*E notation*) to represent floating-point values. In E notation, the number 47281.97 would be 0.4728197E5. The part of the number before the E is the mantissa, and the part after the E is the exponent. In C++, the keyword **`float`** is used to denote such numbers. GCC allows 4 bytes of memory for numbers belonging to `float` data type. The numbers of this data type has normally a precision of 7 digits.

`double` data type (*for double precision floating point numbers*)

In some cases, floating point numbers require more precision. Such numbers are represented by **`double`** data type. The range of numbers that can be handled by `float` type is extended by this data type, because it consumes double the size of `float` data type. In C++, it is assured that the range and precision of `double` will be at least as big as `float`. GCC reserves 8 bytes for storing a double precision value. The precision of `double` data type is generally 15 digits.

`void` data type (*for null or empty set of values*)

The data type **void** is a keyword and it indicates an empty set of data. Obviously it does not require any memory space. The use of this data type will be discussed in detail in Chapter 10.

The size of fundamental data types decreases in the order `double`, `float`, `int` and `char`.

6.4 Type modifiers

Have you ever seen travel bags that can alter its size/volume to include extra bit of luggage? Usually we don't use that extra space. But the zipper attached with the bag helps us to alter its volume either by increasing it or by decreasing. In C++ too, we need data types that can accommodate data of slightly bigger/smaller size. C++ provides **data type modifiers** which help us to alter the size, range or precision. Modifiers precede the data type name in the variable declaration. It alters the range of values permitted to a data type by altering the memory size and sign of values. Important modifiers are **`signed`**, **`unsigned`**, **`long`** and **`short`**.

The exact sizes of these data types depend on the compiler and computer you are using. It is guaranteed that:

- a `double` is at least as big as a `float`.
- a `long double` is at least as big as a `double`.

Each type and their modifiers are listed in Table 6.1 (based on GCC compiler) with their features.

Name	Description	Size	Range
`char`	Character	1 byte	`signed`: -128 to 127 `unsigned`: 0 to 255
`short int` (`short`)	Short Integer	2 bytes	`signed`: -32768 to 32767 `unsigned`: 0 to 65535
`int`	Integer	4 bytes	`signed`: -2147483648 to 2147483647 `unsigned`: 0 to 4294967295
`long int` (`long`)	Long integer	4 bytes	`signed`: -2147483648 to 2147483647 `unsigned`: 0 to 4294967295
`float`	Floating point number	4 bytes	$-3.4\times10^{+/-38}$ to $+3.4 \times 10^{+/-38}$ with approximately 7 significant digits
`double`	Double precision floating point number	8 bytes	$-1.7 \times 10^{+/-308}$ to $+1.7 \times 10^{+/-308}$ with approximately 15 significant digits
`long double`	Long double precision floating point number	12 bytes	$-3.4 \times 10^{+/-4932}$ to $+3.4 \times10^{+/-4932}$ With approximately 19 significant digits

Table 6.1: Data type and type modifiers

The values listed in Table 6.1 are only sample values to give you a general idea of how the types differ. The values for any of these entries may be different on your system.

6.5 Variables

Memory locations are to be identified to refer data. **Variables** are the names given to memory locations. These are identifiers of C++ by which memory locations are referenced to store or retrieve data. The size and nature of data stored in a variable depends on the data type used to declare it. There are three important aspects for a variable.

i. Variable name

It is a symbolic name (identifier) given to the memory location through which the content of the location is referred to.

ii. Memory address

The RAM of a computer consists of collection of cells each of which can store one byte of data. Every cell (or byte) in RAM will be assigned a unique address to refer it. All the variables are connected to one or more memory locations in RAM. The base address of a variable is the starting address of the allocated memory space. In the normal situation, the address is given implicitly by the compiler. The address is also called the L-value of a variable. In Figure 6.2 the base address of the variable **Num** is 1001.

iii. Content

The value stored in the location is called the content of the variable. This is also called the R-value of the variable. Type and size of the content depends on the data type of the variable.

Figure 6.2, shows the memory representation of a variable. Here the variable name is Num and it consumes 4 bytes of memory at memory addresses 1001, 1002, 1003 and 1004. The content of this variable is 18. That is the *L-value* of Num is 1001 and the *R-value* is 18.

Fig. 6.2 : Memory representation of a Variable

6.6 Operators

Operators are tokens constituted by predefined symbols that trigger computer to carry out operations. The participants of an operation are called **operands**. An operand may be either a constant or a variable.

For example, a+b triggers an arithmetic operation in which + (addition) is the operator and a, b are operands. Operators in C++ are classified based on various criteria. Based on number of operands required for the operation, operators are classified into three. They are unary, binary and ternary.

Unary operators

A unary operator operates on a single operand. Commonly used unary operators are unary+ (positive) and unary– (negative). These are used to represent the sign of a number. If we apply unary+ operator on a signed number, the existing sign will not change. If we apply unary– operator on a signed number, the sign of the existing

number will be negated. Examples of the use of unary operators are given in Table 6.2.

Variable x	Unary + +x	Unary- -x
8	8	-8
0	0	0
-9	-9	9

Table 6.2 : Unary operators

Some other examples of unary operators are increment (**++**) and decrement (**--**) operators.

Binary operators

Binary operators operate on two operands. Arithmetic operators, relational operators, logical operators, etc. are commonly used binary operators.

Ternary operator

Ternary operator operates on three operands. The typical example is the conditional operator (? **:**).

The operations triggered by the operators mentioned above will be discussed in detail in the coming sections and some of them will be dealt with in Chapter 7.

Based on the nature of operation, operators are classified into arithmetic, relational, logical, input/output, assignment, short-hand, increment/decrement, etc.

6.6.1 Arithmetic operators

Arithmetic operators are defined to perform basic arithmetic operations such as addition, subtraction, multiplication and division. The symbols used for this are **+**, **–** ,***** and **/** respectively. C++ also provides a special operator, **%** (modulus operator) for getting remainder during division. All these operators are binary operators. Note that **+** and **–** are used as unary operators too. The operands required for these operations are numeric data. The result of these operations will also be numeric. Table 6.3 shows some examples of binary arithmetic operations.

Variable x	Variable y	Addition x + y	Subtraction x - y	Multiplication x * y	Division x / y
10	5	15	5	50	2
-11	3	-8	-14	-33	-3.66667
11	-3	8	14	-33	-3.66667
-50	-10	-60	-40	500	5

Table 6.3 : Arithmetic operators

Modulus operator (%)

The modulus operator, also called as mod operator, gives the remainder value during arithmetic division. This operator can only be applied over integer operands.

Table 6.4 shows some examples of modulus operation. Note that the sign of the result is the sign of the first operand. Here in the table the first operand is **x**.

Variable x	Variable y	Modulus Operation x % y	Variable x	Variable y	Modulus Operation x % y
10	5	0	100	100	0
5	10	5	32	11	10
-5	11	-5	11	-5	1
5	-11	5	-11	5	-1
-11	-5	-1	-5	-11	-5

Table 6.4 : Operations using Modulus operator

Check yourself

1. Arrange the fundamental data types in ascending order of size.
2. The name given to a storage location is known as ___________.
3. Name a ternary operator in C++.
4. Predict the output of the following operations if x = -5 and y = 3 initially:

a. –x	f. x + y
b. –y	g. x % y
c. –x + -y	h. x / y
d. –x – y	i. x * –y
e. x % –11	j. –x % -5

6.6.2 Relational operators

Relational operators are used for comparing numeric data. These are binary operators. The result of any relational operation will be either **True** or **False.** In C++, True is represented by **1** and False is represented by **0.** There are six relational operators in C++. They are **<** (*less than*), **>** (*greater than*), **<=** (*less than or equal to*), **>=** (*greater than or equal to*), **==** (*equal to*) and **!=** (*not equal to*). Note that equality checking requires two equal symbols (**==**). Some examples for the use of various relational operators and their results are shown in Table 6.5.

m	n	m<n	m>n	m<=n	m>=n	m!=n	m==n
12	5	0	1	0	1	1	0
-7	2	1	0	1	0	1	0
4	4	0	0	1	1	0	1

Table 6.5 : Operations using Relational operators

6.6.3 Logical operators

Using relational operators, we can compare values. Examples are 3<5, num!=10, etc. These comparison operations are called relational expressions in C++. In some cases, two or more comparisons may need to be combined. In Mathematics we may use expressions like a>b>c. But in C++ it is not possible. We have to separate this into two, as a>b and b>c and these are to be combined using the logical operator **&&**, i.e. (a>b)&&(b>c). The result of such logical combinations will also be either True or False (i.e. 1 or 0). The logical operators are **&&** (logical AND), **||** (logical OR) and **!** (logical NOT).

Logical AND (&&) operator

If two relational expressions El and E2 are combined using logical AND (**&&**) operator, the result will be 1 (True) only if both E1 and E2 have values 1 (True). In all other cases the result will be 0 (False). The results of evaluation of **&&** operation for different possible combination of inputs are shown in Table 6.6.

E1	E2	E1&&E2
0	0	0
0	1	0
1	0	0
1	1	1

Table 6.6 : Logical AND

Examples: 10>5 && 15<25 evaluates to 1 (True)

10>5 && 100<25 evaluates to 0 (False)

Logical OR (||) operator

If two relational expressions El and E2 are combined using logical OR (**||**) operator, the result will be 0 (False) only if both El and E2 are having value 0 (False). In all other cases the result will be 1 (True). The results of evaluation of **||** operation for different possible combination of inputs are shown in Table 6.7.

<table>
<tr><th>E1</th><th>E2</th><th>E1||E2</th></tr>
<tr><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>1</td><td>1</td></tr>
</table>

Table 6.7 : Logical OR

Examples: 10>5 || 100<25 evaluates to 1(True)

10> 15 || 100<90 evaluates to 0 (False)

Logical NOT operator (!)

This operator is used to negate the result of a relational expression. This is a unary operation. The results of evaluation of **!** operator for different possible inputs are shown in Table 6.8.

E1	!E1
0	1
1	0

Table 6.8 : Logical NOT

Example: !(100<2) evaluates to 1 (True)

!(100>2) evaluates to 0 (False)

6.6.4 Input / Output operators

Usually input operation requires user's intervention. In the process of input operation, the data given through the keyboard is stored in a memory location. C++ provides **>>** operator for this operation. This operator is known as ***get from*** or ***extraction*** operator. This symbol is constituted by two greater than symbols.

Similarly in output operation, data is transferred from RAM to an output device. Usually the monitor is the standard output device to get the results directly. The operator **<<** is used for output operation and is called ***put to*** or ***insertion*** operator. It is constituted by two less than symbols.

6.6.5 Assignment operator (=)

When we have to store a value in a memory location, assignment operator (**=**) is used. This is a binary operator and hence two operands are required. The first operand should be a variable where the value of the second operand is to be stored. Some examples are shown in table 6.9.

Item	Description
a=b	The value of variable **b** is stored in **a**
a=3	The constant **3** is stored in variable **a**

Table 6.9 : Assignment operator

We discussed the usage of the relational operator **==** in Section 6.6.2. See the difference between these two operators. The **=** symbol assigns a value to a variable, whereas **==** symbol compares two values and gives True or False as the result.

6.6.6 Arithmetic assignment operators

A simple arithmetic statement can be expressed in a more condensed form using arithmetic assignment operators. For example, `a=a+10` can be represented as **a+=10**. Here **+=** is an arithmetic assignment operator. This method is applicable

to all arithmetic operators and they are shown in Table 6.10. The arithmetic assignment operators in C++ are **+=**, **-=**, ***=**, **/=**, **%=**. These are also known as C++ short-hands. These are all binary operators and the first operand should be a variable. The use of these operators makes the two operations (arithmetic and assignment) faster than the usual method.

Arithmetic assigment operation	Equivalent arithmetic operation
`x += 10`	`x = x + 10`
`x -= 10`	`x = x - 10`
`x *= 10`	`x = x * 10`
`x /= 10`	`x = x / 10`
`x %= 10`	`x = x % 10`

Table 6.10 : C++ short hands

6.6.7 Increment (++) and Decrement (−−) operators

Increment and decrement operators are two special operators in C++. These are unary operators and the operand should be a variable. These operators help keeping the source code compact.

Increment operator (++)

This operator is used for incrementing the content of an integer variable by one. This can be written in two ways: **++x** (pre increment) and **x++** (post increment). Both are equivalent to `x=x+1` as well as `x+=1`.

Decrement operator (−−)

As a counterpart of increment operator, there is a decrement operator which decrements the content of an integer variable by one. This operator is also used in two ways: **--x** (pre decrement) and **x--** (post decrement). These are equivalent to `x=x-1` and `x-=1`.

The two usages of these operators are called prefix form and postfix form of increment/decrement operation. Both the forms make the same effect on the operand variable, but the mode of operation will be different when these are used with other operators.

Prefix form of increment/decrement operators

In the prefix form, the operator is placed before the operand and the increment/decrement operation is carried out first. The incremented/decremented value is used for the other operations. So, this method is often called ***change, then use*** method.

Consider the variables `a`, `b`, `c` and `d` with values `a=10`, `b=5`. If an operation is specified as `c=++a`, the value of `a` will be 11 and that of `c` will also be 11. Here the value of `a` is incremented by 1 at first and then the changed value of `a` is assigned

to c. That is why both the variables get the same value. Similarly, after the execution of `d=--b` the value of `d` and `b` will be 4.

Postfix form of increment/decrement operators

When increment/decrement operation is performed in postfix form, the operator is placed after the operand. The current value of the variable is used for the remaining operations and after that the increment/decrement operation is carried out. So, this method is often called ***use, then change*** method.

Consider the same variables used above with the same initial values. After the operation performed with `c=a++`, the value of `a` will be 11, but that of `c` will be 10. Here the value of `a` is assigned to `c` at first and then `a` is incremented by 1. That is, before changing the value of `a` it is used to assign to `c`. Similarly, after the execution of `d=b--` the value of `d` will be 5 but that of `b` will be 4.

6.6.8 Conditional operator (? :)

This is a ternary operator applied over three operands. The first operand will be a logical expression (condition) and the remaining two are values. They can be constants, variables or expressions. The condition will be checked first and if it is True, the second operand will be selected to get the value, otherwise the third operand will be selected. Its syntax is:

```
Expression1? Expression2: Expression3
```

Let us see the operation in the following:

```
result = score>50 ? 'p' : 'f'
```

If the value of `score` is greater than 50 then the value `'p'` is assigned to the variable `result`, else value `'f'` is assigned to `result`. More about this operator will be discussed in Chapter 7.

6.6.9 `sizeof` operator

The operator `sizeof` is a unary compile-time operator that returns the amount of memory space in bytes allocated for the operand. The operand can be a constant, a variable or a data type. The syntax followed is given below:

- `sizeof (data_type)`
- `sizeof variable_name`
- `sizeof constant`

It is to be noted that when data type is used as the operand for `sizeof` operator, it should be given within a pair of parentheses. For the other operands parentheses are not compulsory. Table 6.11 shows different forms of usages of `sizeof` operator.

Item	Description
`sizeof(int)`	Gives the value 4 (In GCC, size of `int` data type is 4 bytes)
`sizeof 3.2`	Returns 8 (A floating point constant will be taken as `double` type data)
`sizeof p;`	If `p` is `float` type variable, it gives the value 4.

Table 6.11: Various usages of sizeof operator

6.6.10 Precedence of operators

Let us consider the case where different operators are used with the required operands. We should know in which order the operations will be carried out. C++ gives priority to the operators for execution. During evaluation, pair of parentheses is given the first priority. If the expression is not parenthesised, it is evaluated according to the predefined precedence order. The order of precedence for the operators is given in Table 6.12. In an expression, if the operators of the same priority level occur, the precedence of execution will be from left to right in most of the cases.

Priority	Operations
1	`( )` parentheses
2	`++`, `--`, `!` , Unary+ , Unary –, `sizeof`
3	* (multiplication), / (division), % (Modulus)
4	+ (addition), – (subtraction)
5	< (less than), <= (less than or equal to), > (greater than), >= (greater than or equal to)
6	== (equal to), != (not equal to)
7	`&&` (logical AND)
8	`||` (logical OR)
9	`? :` (Conditional expression)
10	= (Assignment operator), *=, /=, %=, +=, –= (arithmetic assignment operators)
11	, (Comma)

Table 6.12: Precedence of operators

Consider the variables with values: `a=3, b=5, c=4, d=2, x`

After the operations specified in `x = a + b * c - d`, the value in `x` will be 21. Here * (multiplication) has higher priority than + (addition) and – (subtraction). Therefore the variables `b` and `c` are multiplied, then that result is added to `a`. From that result, `d` is subtracted to get the final result.

It is important to note that the operator priority can be changed in an expression as per the need of the programmer by using parentheses ( ). For example, if `a=5`, `b=4`, `c=3`, `d=2` then the result of `a+b-c*d` will be 3. Suppose the programmer wants to perform subtraction first and then the addition and multiplication, you need to use proper parentheses as `(a+(b-c))*d`. Now the output will be 12. For changing operator priority, brackets `[]` and braces `{}` cannot be used.

The operator precedence may be different for different types of compilers. Turbo C++ gives higher precedence to prefix increment / decrement than its postfix form.

For example, if a is initially 5, the values of b and a after b=a++ + ++a are 12 and 7 respectively. This is equivalent to the set of statements a=a+1 (prefix expansion), b=a+a, and a=a+1 (postfix expansion).

6.7 Expressions

An expression is composed of operators and operands. The operands may be either constants or variables. All expressions can be evaluated to get a result. This result is known as the value returned by the expression. On the basis of the operators used, expressions are mainly classified into arithmetic expressions, relational expressions and logical expressions.

6.7.1 Arithmetic expressions

An expression in which only arithmetic operators are used is called arithmetic expression. The operands are numeric data and they may be variables or constants. The value returned by these expressions is also numeric. Arithmetic expressions are further classified into integer expressions, floating point (real) expressions and constant expressions.

Integer expressions

If an arithmetic expression contains only integer operands, it is called integer expression and it produces an integer result after performing all the operations given in the expression. For example, if `x` and `y` are integer variables, some integer expressions and their results are shown in Table 6.13. Note that all the above expressions produce integer values as the results.

x	y	x + y	x / y	-x + x * y	5 + x / y	x % y
5	2	7	2	5	7	1
6	3	9	2	12	7	0

Table 6.13: Integer expressions and their results

Floating point expressions (Real expressions)

An arithmetic expression that is composed of only floating point data is called floating point or real expression and it returns a floating point result after performing all the operations given in the expression. Table 6.14 shows some real expressions and their results, assuming that `x` and `y` are floating point variables.

x	y	x + y	x / y	-x + x * y	5 + x / y	x * x / y
5.0	2.0	7.0	2.5	5.0	7.5	12.5
6.0	3.0	9.0	2.0	12.0	7.0	12.0

Table 6.14: Floating point expressions and their results

It can be seen that all the above expressions produce floating point values as the results.

In an arithmetic expression, if all the operands are constant values, then it is called **constant expression**. The expression 20+5/2.0 is an example. The constants like `15, 3.14, 'A'` are also known as constant expressions.

6.7.2 Relational expressions

When relational operators are used in an expression, it is called relational expression and it produces Boolean type results like True (1) or False (0). In these expressions, the operands are numeric data. Let us see some examples of relational expressions in Table 6.15.

x	y	x > y	x == y	x+y !=y	x-2 == y+1	x*y == 6*y
5.0	2.0	1 (True)	0 (False)	1 (True)	1 (True)	0 (False)
6	13	0 (False)	0 (False)	1 (True)	0 (False)	1 (True)

Table 6.15: Relational expressions and their results

We know that arithmetic operators have higher priority than relational operators. So when arithmetic expressions are used on either side of a relational operator, arithmetic operations will be carried out first and then the results are compared. The table contains some expressions in which both arithmetic and relational operators are involved. Though they contain mixed type of operators, they are called relational expressions since the final result will be either True or False.

6.7.3 Logical expressions

Logical expressions combine two or more relational expressions with logical operators and produce either True or False as the result. A logical expression may contain constants, variables, logical operators and relational operators. Let us see some examples in Table 6.16.

x	y	x>=y && x==20	x==5||y==0	x==y && y+2==0	!(x==y)
5.0	2.0	0 (False)	1 (True)	0 (False)	1 (True)
20	13	1 (True)	0 (False)	0 (False)	1 (True)

Table 6.16: Logical expressions and their results

As seen in Table 6.16, though some expressions consist of arithmetic and relational operators in addition to logical operators, the expressions are considered as logical expressions. This is because the operation carried out at last will be the logical operation and the result will be either True or False.

Check yourself

1. Predict the output of the following operations if x=5 and y=3.
   a. x>=10 && y>=4
   b. x>=1 && y>=3
   c. x>=1 || y>=4
   d. x>=1 || y>=3
2. Predict the output if p=5, q=3, r=2
   a. ++p-q*r/2
   b. p*q--+r
   c. p-q-r*2+p
   d. p+=5*q+r*r/2

6.8 Type conversion

As discussed earlier arithmetic expressions are of two types, integer expressions and real expressions. In both cases, the operands involved in the arithmetic operation are of the same data type. But there are situations where different types of numeric data may be involved. For example in C++, the integer expression 5 /2 gives 2 and the real expression 5.0/2.0 gives 2.5. But what will the result of 5/2.0 or 5.0/2 be? Conversion techniques are applied in such situations. The data type of one operand will be converted to another. It is called **type conversion** and can be done in two ways: implicitly and explicitly.

6.8.1 Implicit type conversion (Type promotion)

Implicit type conversion is performed by C++ compiler internally. In expressions where different types of data are involved, C++ converts the lower sized operands to the data type of highest sized operand. Since the conversion is always from lower type to higher, it is also known as **type promotion**. Data types in the decreasing order of size are as follows: `long double`, `double`, `float`, `unsigned long`, `long int and unsigned int / short int`. The type of the result will also be the type of the highest sized operand.

For example, the expression 5 / 2 * 3 + 2.5 gives the result 8.5. The evaluation steps are as follows:

Step 1: 5 / 2 → 2 (Integer division)

Step 2: 2 * 3 → 6 (Integer multiplication)

Step 3: 6 + 2.5 → 8.5 (Floating point addition, 6 is converted into 6.0)

6.8.2 Explicit type conversion (Type casting)

Unlike implicit type conversion, sometimes the programmer may decide the data type of the result of evaluation. This is done by the programmer by specifying the data type within parentheses to the left of the operand. Since the programmer explicitly casts a data to the desired type, it is known as explicit type conversion or **type casting**. Usually, type casting is applied on the variables in the expressions. More examples will be discussed in Section 6.9.2.

6.9 Statements

Can you recollect the learning hierarchy of a natural language? Alphabet, words, phrases, sentences, paragraphs and so on. In the learning process of C++ language we have covered character set, tokens and expressions. Now we have come to the stage where we start communication with the computer sensibly and meaningfully with the help of statements. **Statements** are the smallest executable unit of a programming language. C++ uses the symbol semicolon (`;`) as the delimiter of a statement. Different types of statements used in C++ are declaration statements, assignment statements, input statements, output statements, control statements etc. Each statement has its own purpose in a C++ program. All these statements except declaration statements are executable statements as they possess some operations to be done by the computer. Executable statements are the instructions to the computer. The execution of control statements will be discussed in Chapter 7. Let us discuss the other statements.

6.9.1 Declaration statements

Every user-defined word should be defined in the program before it is used. We have seen that a variable is a user-defined word and it is an identifier of a memory location. It must be declared in the program before its use. When we declare a variable, we tell the compiler about the type of data that will be stored in it. The syntax of variable declaration is:

```
data_type <variable1>[, <variable2>, <variable3>,...];
```

The `data_type` in the syntax should be any valid data type of C++. The syntax shows that when there are more than one variables in the declaration, they are separated by comma. The declaration statement ends with a semicolon. Typically, variables are declared either just before they are used or at the beginning of the

program. In the syntax, everything given inside the symbols [and] are optional. The following statements are examples for variable declaration:

```
int rollnumber;
double wgpa, avg_score;
```

The first statement declares the variable `rollnumber` as **`int`** type so that it will be allocated four bytes of memory (*as per GCC*) and it can hold an integer number within the range from -2147483648 to +2147483647. The second statement defines the identifiers `wgpa` and `avg_score` as variables to hold data of **`double`** type. Each of them will be allocated 8 bytes of memory. The memory is allocated to the variables during the compilation of the program.

Variable initialisation

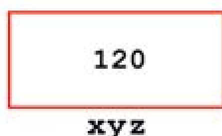
We saw, in Section 6.5, that a variable is associated with two values: L-value (its address) and R-value (its content). When a variable is declared, a memory location with an address will be allocated for it. What will its content be? It is not blank or 0 or space! If the variable is declared with `int` data type, the content or R-value will be any integer within the allowed range. But this number cannot be predicted or will not always be the same. So we call it *garbage value*. When we store a value into the variable, the existing content will be replaced by the new one. The value can be stored in the variable either at the time of compilation or execution. Supplying value to a variable at the time of its declaration is called **variable initialisation**. This value will be stored in the respective memory location during compile-time. The assignment operator (=) is used for this. It can be done in two ways as given below:

```
data_type variable = value;
```

OR

```
data_type variable(value)
```

The statements: `int xyz =120;` and `int xyz(120);` are examples of variable initialisation statements. Both of these statements declare an integer variable `xyz` and store the value 120 in it as shown in Figure 6.3.



xyz

Fig. 6.3: Variable initialisation

More examples are:

```
float val=0.12, b=5.234;
char k='A';
```

A variable can also be initialised during the execution of the program and is known as dynamic initialisation. This is done by assigning an expression to a variable as shown in the following statements:

```
float product =  x * y;
float interest = p*n*r/100.0;
```

In the first statement, the variable `product` is initialised with the product of the values stored in `x` and `y` at runtime. In the second case, the expression `p*n*r/100.0` is evaluated and the value returned by it will be stored in the variable `interest`.

Note that during dynamic initialisation, the variables included in the expression at the right of assignment operator should have valid data. Otherwise it will produce unexpected results.

`const` – The access modifier

It is a good practice to use symbolic constants rather than using numeric constants directly. For example, we can use symbolic names like Pi instead of using 22.0/7.0 or 3.14. The keyword **`const`** is used to create such symbolic constants whose value can never be changed during execution. Consider the following statement:

```
float pi=3.14;
```

The floating point variable **`pi`** is initialised with the value 3.14. The content of **`pi`** can be changed during the execution of the program. But if we modify the declaration as:

```
const float pi=3.14;
```

the value of **`pi`** remains constant (unaltered) throughout the execution of the program. The read/write accessibility of the variable is modified as read only. Thus, the **`const`** acts as an access modifier.

During software development, larger programs are developed using collaborative effort. Several people may work together on different portions of the same program. They may share the same variable. In these situations, there may be occasions where one may modify the content of the variable which will adversely affect other person's coding. In these situations we have to keep the content of variables unaffected by the activity of others. This can be done by using 'const'.

6.9.2 Assignment statements

When the assignment operator (=) is used to assign a value to a variable, it forms an assignment statement. It can take any of the following syntax:

```
variable  =  constant;
variable1  =  variable2;
variable  =  expression;
variable  =  function();
```

In the third case, the result of the expression is stored in the variable. Similarly, in the fourth case, the value returned by the function is stored. The concept of functions will be discussed in Chapter 10.

Some examples of assignment statements are given below:

```
A = 15;                          b = 5.8;
c = a + b;                       c = a * b;
d = (a + b)*(c + d);             r = sqrt(25);
```

In the last example, `sqrt()` is a function that assigns the square root of 25 to the variable `r`.

The left hand side (LHS) of an assignment statement must be a variable. During execution, the expression at the right hand side (RHS) is evaluated first. The result is then assigned (stored) to the variable at LHS.

Assignment statement can be chained for doing multiple assignments at a time. For instance, the statement `x=y=z=13;` assigns the value 13 in three variables in the order of `z`, `y` and `x`. The variables should be declared before this assignment. If we assign a value to a variable, the previous value in it, if any, will be replaced by the new value.

Type compatibility

During the execution of an assignment statement, if the data type of the RHS expression is different from that of the LHS variable, there are two possibilities.

- The size of the data type of the variable at LHS is higher than that of the variable or expression at RHS. In this case data type of the value at RHS is promoted (type promotion) to that of the variable at LHS. Consider the following code snippet:

  ```
  int a=5, b=2;
  float p, q;
  p = b;
  q = a / p;
  ```

  Here the data type of `b` is promoted to `float` and 2.0 is stored in `p`. When the expression `a/p` is evaluated, the result will be 2.5 due to the type promotion of `a`. So, `q` will be assigned with 2.5.

- The second possibility is that the size of the data type of LHS variable is smaller than the size of RHS value. In this case, the higher order bits of the result will be truncated to fit in the variable location of LHS. The following code illustrates this.

  ```
  float a=2.6;
  int p, q;
  p = a;
  q = a * 4;
  ```

  Here the value of `p` will be 2 and that of `q` will be 10. The expression `a*4` is evaluated to 10.4, but `q` being `int` type it will hold only 10.

Programmer can apply the explicit conversion technique to get the desired results when there are mismatches in the data types of operands. Consider the following code segment.

```
int p=5, q=2;
float x, y;
x = p/q;
y = (x+p)/q;
```

After executing the above code, the value of `x` will be 2.0 and that of `y` will be 3.5. The expression `p/q` being an integer expression gives 2 as the result and is stored in `x` as floating point value. In the last statement, the pair of parentheses gives priority to `x+p` and the result will be 7.0 due to the type promotion of `p`. Then the result 7.0 will be the first operand for the division operation and hence the result will be 3.5 since `q` is converted into `float`. If we have to get the floating point result from `p/q` to store in `x`, the statement should be modified as `x=(float)p/q;` or `x=p/(float)q;` by applying type casting.

6.9.3 Input statements

Input statement is a means that allows the user to store data in the memory during the execution of the program. We saw that the ***get from*** or ***extraction*** operator (**>>**) specifies the input operation. The operands required for this operator are the input device and a location in RAM where data is to be stored. Keyboard being a standard console device, the stream (sequence) of data is extracted from the keyboard and stored in memory locations identified by variables. Since C++ is an object oriented language, keyboard is considered as the standard input stream device and is identified as an object by the name **`cin`** (pronounced as 'see in'). The simplest form of an input statement is:

```
streamobject >> variable;
```

Since we use keyboard as the input device, the `streamobject` in the syntax will be substituted by `cin`. The operand after the **>>** operator should strictly be a variable. For example, the following statement reads data from the keyboard and stores in the variable **`num`**.

```
cin >> num;
```

Figure 6.4 shows how data is extracted from keyboard and stored in the variable.

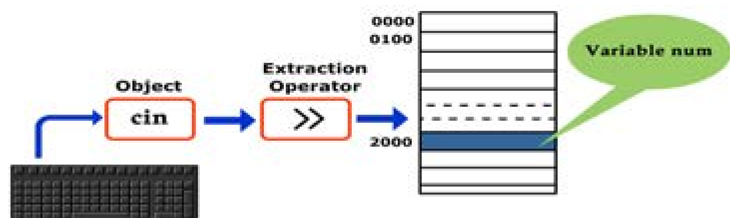


Fig 6.4 : Input procedure in C++

6.9.2 Output statements

Output statements make the results available to users through any output device. The ***put to*** or ***insertion*** operator (**<<**) is used to specify this operation. The operands in this case are the output device and the data for the output. The syntax of an output statement is:

```
streamobject << data;
```

The `streamobject` may be any output device and the data may be a constant, a variable or an expression. We use monitor as the commonly used output device and C++ identifies it as an object by the name **`cout`** (pronounced as 'see out'). The following are some examples of output statement with monitor as the output device:

```
cout << num;
cout << "hello friends";
cout << num+12;
```

The first statement displays the content of the variable **`num`**. The second statement displays the string constant `"hello friends"` and the last statement shows the value returned by the expression `num+12` (*assuming that* `num` *contains numeric value*). Figure 6.5 shows how data is inserted into the output stream object (monitor) from the memory location **`num`**.

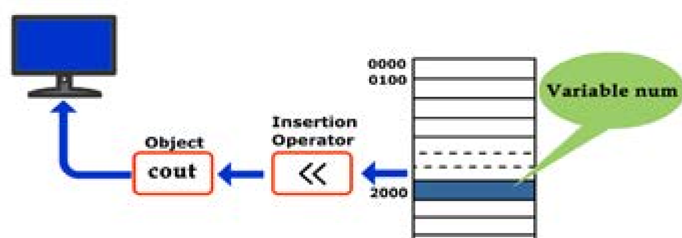


Fig. 6.5: Output procedure in C++

The tokens cin and cout are not keywords. They are predefined words that are not part of the core C++ language, and you are allowed to redefine them. They are defined in libraries required by the C++ language standard. Needless to say, using a predefined identifier for anything other than its standard meaning can be confusing and dangerous and such practice should be avoided. The safest and easiest practice is to treat all predefined identifiers as if they were keywords.

Cascading of I/O operators

Suppose you want to input three values to different variables, say x, y, and z. You may use the following statements:

```
cin>>x;
cin>>y;
cin>>z;
```

But these three statements can be combined to form a single statement as given below:

```
cin>>x>>y>>z;
```

The multiple use of input or output operators in a single statement is called **cascading of I/O operators**. In the use of cascading of input operators, the values input are assigned to the variables from left to right. In the example `cin>>x>>y>>z;` the first value is assigned to `x`, the second to `y` and the third to `z`. While entering values to the variables `x`, `y` and `z` during execution the values should be separated by space bar, tab, or carriage return.

Similarly, if you want to display the contents of different variables (say `x`, `y`, `z`), use the following statement:

```
cout<<x<<y<<z;
```

If variables, constants and expressions appear together for output operations, the above technique can be applied as in the following example:

```
cout<<"The number is "<<z;
```

While cascading output operators, the values for the output will be retrieved from right to left. Consider the code fragment given below:

```
int x=5;
cout<<x<<'\t'<<++x;
```

The output of this code will be: 6 6

It will not be: 5 6

It is to be noted that both **<<** and **>>** operators cannot be used in a single statement.

In the statement `x=y=z=5;` the **=** operator is cascaded. Here also the cascading is from right to left.

6.10 Structure of a C++ program

We are now in a position to solve simple problems by using the statements we discussed so far. But a set of statements alone does not constitute a program. A C++ program has a typical structure. It is a collection of one or more functions. A function means the set of instructions to perform a particular task referred to by a name. Since there can be many functions in a C++ program, they are usually identified by unique names. The most essential function needed for every C++ program is the **`main()`** function.

The structure of a simple C+ + program is given below:

```
#include <header file>
using namespace identifier;
int main()
{
    statements;
       :
       :
       :
    return 0;
}
```

The first line is called preprocessor directive and the second line is the namespace statement. The third line is the function header which is followed by a set of statements enclosed by a pair of braces. Let us discuss each of these parts of the program.

6.10.1 Preprocessor directives

A C++ program starts with pre-processor directives. Preprocessors are the compiler directive statements which give instruction to the compiler to process the information provided before actual compilation starts. Preprocessor directives are lines included in the code that are not program statements. These lines always start with a **#** (hash) symbol. The pre-processor directive `#include` is used to link the header files available in the C++ library by which the facilities required in the program can be obtained. No semicolon (**;**) is needed at the end of such lines. Separate `#include` statements should be used for different header files. There are some other pre-processor directives such as `#define`, `#undef`, etc.

6.10.2 Header files

Header files contain the information about functions, objects and predefined derived data types and they are available along with compiler. There are a number of such files to support C++ programs and they are kept in the standard library. Whichever program requires the support of any of these resources, the concerned header file is to be included. For example, if we want to use the predefined objects `cin` and `cout`, we have to use the following statement at the beginning of the program.

```
#include <iostream>
```

The header file `iostream` contains the information about the objects `cin` and `cout`. Eventhough header files have the extension `.h`, it should not be specified for GCC. But the extension is essential for some other compilers like Turbo C++ IDE.

6.10.3 Concept of namespace

A program cannot have the same name for more than one identifier (variables or functions) in the same scope. For example, in our home two or more persons (or even living beings) will not have the same name. If there are, it will surely make conflicts in the identity within the home. So, within the scope of our home, a name should be unique. But our neighbouring home may have a person (or any living being) with the same name as that of one of us. It will not make any confusion of identity within the respective scopes. But an outsider cannot access a particular person by simply using the name; but the house name is also to be mentioned.

The concept of namespace is similar to a house name. Different identifiers are associated to a particular namespace. It is actually a group name in which each item is unique in its name. User is allowed to create own namespaces for variables and functions. We can use an identifier to give name to a namespace. The keyword `using` technically tells the compiler about a namespace where it should search for the elements used in the program. In C++, `std` is an abbreviation of '`standard`' and it is the standard namespace in which `cout`, `cin` and a lot of other objects are defined. So, when we want to use them in a program, we need to follow the format `std::cout` and `std::cin`. This kind of explicit referencing can be avoided with the statement `using namespace std;` in the program. In such a case, the compiler searches this namespace for the elements `cin`, `cout`, `endl`, etc. So whenever the computer comes across `cin`, `cout`, `endl` or anything of that matter in the program, it will read it as `std::cout`, `std::cin` or `std::endl`.

The statement `using namespace std;` doesn't really add a function, it is the `#include <iostream>` that "loads" `cin`, `cout`, `endl` and all the like.

6.10.4 The `main()` function

Every C++ program consists of a function named `main()`. The execution starts at `main()` and ends within `main()`. If we use any other function in the program, it is called (or invoked) from `main()`. Usually a data type precedes the `main()` and in GCC, it should be `int`.

The function header `main()` is followed by its body, which is a set of one or more statements within a pair of braces `{ }`. This structure is known as the definition of the `main()` function. Each statement is delimited by a semicolon (`;`). The statements may be executable and non-executable. The executable statements represent instructions to be carried out by the computer. The non-executable statements are intended for compiler or programmer. They are informative statements. The last statement in the body of `main()` is return 0;. Even though we do not use this statement, it will not make any error. Its relavance will be discussed in Chapter 10.

C++ is a free form language in the sense that it is not necessary to write each statement in new lines. Also a single statement can take more than one line.

6.10.5 A sample program

Let us look at a complete program and familiarise ourselves with its features, in detail. This program on execution will display a text on the screen.

```
#include <iostream>
using namespace std;
int main()
{
      cout<<"Hello, Welcome to C++";
      return 0;
}
```

The program has seven lines as detailed below:

Line 1: The preprocessor directive `#include` is used to link the header file `iostream` with the program.

Line 2: The `using namespace` statement makes available the identifier `cout` in the program.

Line 3: The header of the essential function for a C++ program, i.e., `int main()`.

Line 4: An opening brace { that marks the beginning of the instruction set (program).

Line 5: An output statement, which will be executed when we run the program, to display the text "`Hello, Welcome to C++`" on the monitor. The header file `iostream` is included in this program to use `cout` in this statement.

Line 6: The `return` statement stops the execution of the `main()` function. This statement is optional as far as `main()` is concerned.

Line 7: A closing brace } that marks the end of the program.

6.11 Guidelines for coding

A source code looks good when the coding is legible, logic is communicative and errors if any are easily detectable. These features can be experienced if certain styles are followed while writing programs. Some guidelines are discussed in this section to write stylistic programs.

Use suitable naming convention for identifiers

Suppose we have to calculate the salary for an employee after deductions. We may code it as: `A = B - C;`

where `A` is the net salary, `B` the total salary and `C` total deduction. The variable names `A`, `B` and `C` do not reflect the quantities they denote. If the same instruction is expressed as follows, it would be better:

```
Net_salary = Gross_salary - Deduction;
```

The variable names used in this case help us to remember the quantity they possess. They readily reflect their purpose. These kinds of identifiers are called ***mnemonic names***. The following points are to be remembered in the choice of names:

- Choose good mnemonic names for all variables, functions and procedures.
  e.g. `avg_hgt`, `Roll_No`, `emp_code`, `SumOfDigits`, etc.
- Use standardized prefixes and suffixes for related variables.
  e.g. `num1`, `num2`, `num3` for three numbers
- Assign names to constants in the beginning of the program.
  e.g. `float PI = 3.14;`

Use clear and simple expressions

Some people have a tendency to reduce the execution time by sacrificing simplicity. This should be avoided. Consider the following example. To find out the remainder after division of `x` by `n`, we can code as: `y = x-(x/n)*n;`

The same thing is achieved by a simpler and more elegant piece of code as shown below:

```
y = x % n;
```

So it is better to use simpler codes in programming to make the program more simple and clear.

Use comments wherever needed

Comments play a very important role as they provide internal documentation of a program. They are lines in code that are added to describe the program. They are ignored by the compiler. There are two ways to write comments in C++:

Single line comment: The characters **//** (two slashes) is used to write single line comments. The text appearing after **//** in a line is treated as a comment by the C++ compiler.

Multiline comments: Anything written within **/*** and ***/** is treated as comment so that the comment can take any number of lines.

But care should be taken that no relevant code of the program is included accidently inside the comment. The following points are to be noted while commenting:

- Always insert prologues, the comments in the beginning of a program that summarises the purpose of the program.
- Comment each variable and constant declaration.
- Use comments to explain complex program steps.
- It is better to include comments while writing the program itself.
- Write short and clear comments.

Relevance of indentation

In computer programming, an indent style is a convention governing the indentation of blocks of code to convey the program's structure, for good visibility and better clarity. An indentation makes the statements clear and readable. It shows the levels of statements in the program.

The usage of these guidelines can be observed in the programs given in the next section.

Program gallery

Let us now write programs to solve some problems following the coding guidelines. The call-outs given are not part of the program. Program 6.1 displays a message.

Program 6.1: To display a message

```
    /* This program displays the message
       "Smoking is injurious to health"
       on the monitor */
#include <iostreamh>  // To use the cout object
using namespace std; // To access cout
int main() //program begins here
{   //The following output statement displays a message
    cout << "Smoking is injurious to health";
    return 0;
}   //end of the program
```

Multiline comment

Single line comment

Indentation

On executing Program 6.1, the output will be as follows:

```
Smoking is injurious to health
```

More illustrations on the usage of indentation can be seen in the examples given in Chapter 7.

Program 6.2 accepts two integer numbers from the user, finds its sum and displays the result.

Program 6.2: To find the sum of two integer numbers

```
#include <iostream>
using namespace std;
int main()
{  //Program begins
/* Two variables are declared to read user inputs and the
variable sum is declared to store the result
*/
   int num1, num2, sum;
   cout<<"Enter two numbers: "; //Prompt for input
   cin>>num1>>num2;    //Cascading to get two numbers
   sum=num1+num2;  //Assignment statement to find the sum
   cout<<"Sum of the entered numbers = "<<sum;
/* The result is displayed with proper message.
   Cascading of output operator is utilized     */
   return 0;
}
```

A sample output of Program 6.2 is given below:

```
Enter two numbers: 5   7
Sum of the entered numbers = 12
```

User inputs separated by spaces

Let us consider another problem. A student is awarded with three scores obtained in three Continuous Evaluation (CE) activities. The maximum score of an activity is 20. Find the average score of the student.

Program 6.3: To find the average of three CE scores

```
#include <iostream>
using namespace std;
int main()
{
    int score_1, score_2, score_3;
    float avg;
    //Average of 3 numbers can be a floating point value
    cout << "Enter the three CE scores: ";
    cin >> score_1 >> score_2 >> score_3;
    avg = (score_1 + score_2 + score_3) / 3.0;
```

```
/* The result of addition will be an integer value. If 3
is written instead of 3.0, integer division will be
performed and will not get the correct result   */
    cout << "Average CE score is: " << avg;
    return 0;
}
```

Program 6.3 gives the following output for the CE scores 17, 19 and 20.

```
Enter the three CE scores: 17   19   20
Average CE score is: 18.666666
```

The assignment statement to find the average value uses an expression to the right of assignment operator (=). This expression has two + operators and one / operator. The precedence of / over + is changed by using parentheses for addition. The operands for the addition operators are all `int` type data and hence the result will be an integer. When this integer result is divided by 3, the output will again be an integer. If it was so, the output of Program 6.3 would have been 18, which is not accurate. Hence floating point constant 3.0 is used as the second operand for / operator. It makes the integer numerator `float` by type promotion.

Suppose the radius of a circle 'r' is given and you are requested to compute its area and the perimeter. As you know, area of a circle is calculated using the formula πr^2 and perimeter by $2\pi r$, where $\pi = 3.14$. Program 6.4 solves this problem.

Program 6.4: To find the area and perimeter of a circle for a given radius

```
#include <iostream>
using namespace std;
int main()
{
    const float PI = 22.0/7; //Use of const access modifier
    float radius, area, perimeter;
    cout<<"Enter the radius of the circle: ";
    cin>>radius;
    area = PI * radius * radius;
    perimeter = 2 * PI * radius;
    cout<<"Area of the circle = "<<area<<"\n";
    cout<<"Perimeter of the circle = "<<perimeter;
    return 0;
}
```

Escape sequence '\n' prints a new line after displaying the value of Area

A sample output of Program 6.4 is as follows:

```
Enter the radius of the circle: 2.5
Area of the circle = 19.642857
Perimeter of the circle = 15.714285
```

The last two output statements of Program 6.4 displays both the results in separate lines. The escape sequence character '\n' brings the cursor to the new line before the last output statement gets executed.

Let us develop another program to find simple interest. As you know, principal amount, rate of interest and period are to be given as input to get the result.

Program 6.5: To find the simple interest

```
#include <iostream>
using namespace std;
int main()
{
    float p_Amount, n_Year, i_Rate, int_Amount;
    cout<<"Enter the principal amount in Rupees: ";
    cin>>p_Amount;
    cout<<"Enter the number of years for the deposit: ";
    cin>>n_Year;
    cout<<"Enter the rate of interest in percentage: ";
    cin>>i_Rate;
    int_Amount = p_Amount * n_Year * i_Rate /100;
    cout << "Simple interest for the principal amount "
         <<p_Amount<<" Rupees for a period of "<<n_Year
         <<" years at the rate of interest "<<i_rate
         <<" is "<<int_Amount<<" Rupees";
    return 0;
}
```

A sample output of Program 6.5 is given below:

```
Enter the principal amount in Rupees: 100
Enter the number of years for the deposit: 2
Enter the rate of interest in percentage: 10
Simple interest for the principal amount 100 Rupees for a
period of 2 years at the rate of interest 10 is 20 Rupees
```

The last statement in Program 6.5 is the output statement and it spans over four lines. Note that there is no semi colon at the end of each line and so it is considered a single statement. On execution of the program the result may be displayed in multiple lines depending on the size and resolution of the monitor of your computer.

Program 6.6 solves a temperature conversion problem. The temperature in degree celsius will be given as input and the output will be its equivalent in fahrenheit.

Program 6.6: To convert temperature from Celsius to Fahrenheit

```
#include <iostream>
using namespace std;
int main()
{
    float celsius, fahrenheit;
    cout<<"Enter the Temperature in Celsius: ";
    cin>>celsius;
    fahrenheit=1.8*celsius+32;
    cout<< celsius<<" Degree Celsius = "
        << fahrenheit<<" Degree Fahrenheit";
    return 0;
}
```

Program 6.6 gives a sample output as follows:

```
Enter the Temperature in Celsius: 37
37 Degree Celsius = 98.599998 Degree Fahrenheit
```

We know that each character literal in C++ has a unique value called its ASCII code. These values are integers. Let us write a program to find the ASCII code of a given character.

Program 6.7: To find the ASCII value of a character

```
#include <iostream>
using namespace std;
int main()
{
    char ch;
    int asc;
    cout << "Enter the character: ";
    cin >> ch;
    asc = ch;
    cout << "ASCII value of "<<ch<<" = " << asc;
    return 0;
}
```

A sampler output of Program 6.7 is given below:

```
Enter the character: A
ASCII value of A = 65
```

Let us sum up

Data types are means to identify the type of data and associated operations handling it. Each data type has a specific size and a range of data. Data types are used to declare variables. Type modifiers help handling a higher range of data and are used with data types to declare variables. Different types of operators are available in C++ for various operations. When operators are combined with operands (data), expressions are formed. There are mainly three types of expressions - arithmetic, relational and logical. Type conversion methods are used to get desired results from arithmetic expressions. Statements are the smallest executable unit of a program. Variable declaration statements define the variables in the program and they will be allocated memory space. The executable statements like assignment statements, input statements, output statements, etc. help giving instructions to the computer. Some special operators like arithmetic assignment, increment, decrement, etc. make the expressions and statements compact and the execution faster. C++ program has a typical structure and it must be followed while writing programs. Stylistic guidelines shall be followed to make the program attractive and communicative among humans.

Learning outcomes

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

- identify the various data types in C++.
- list and choose appropriate data type modifiers.
- choose appropriate variables.
- experiment with various operators.
- apply the various I/O operators.
- write various expressions and statements.
- identify the structure of a simple C++ program.
- identify the need for stylistic guidelines while writing a program.
- write simple programs using C++.

Lab activity

1. Write a program that asks the user to enter the weight in grams, and then display the equivalent in Kilograms.
2. Write a program to generate the following table

2013	100%
2012	99.9%
2011	95.5%
2010	90.81%
2009	85%

   Use a single cout statement for output. (Hint: Make use of \n and \t)

4. Write a short program that asks for your height in Meter and Centimeter and converts it to Feet and inches. (1 foot = 12 inches, 1 inch = 2.54 cm).
5. Write a program to compute simple interest and compound interest.
6. Write a program to : (i) print ASCII code for a given digit, (ii) print ASCII code for backspace. (Hint : Store escape sequence for backspace in an integer variable).
7. Write a program to accept a time in seconds and convert into hrs: mins: secs format. For example, if 3700 seconds is the input, the output should be 1hr: 1 min: 40 secs.

Sample questions

Very short answer type

1. What are data types? List all predefined data types in C++.
2. What is a constant?
3. What is dynamic initialisation of variables?
4. What is type casting?
5. Write the purpose of declaration statement?
6. Name the header file to be included to use cin and cout in programs?
7. What is the input operator ">>" and output operator "<<" called ?
8. What will be the result of a = 5/3 if a is (i) `float` and (ii) `int` ?

9. What will be the value of P= P++ + ++i where **i** is 22 and P= 3 initially?
10. Find the value given by the following expression if j =5 initially.

    (i) (5*++j)%6 (ii) (5*j++)%6
11. What will be the order of evaluation for following expressions?

    (i) i+5>=j-6 (ii) s+10<p-2+2*q
12. What will be the result of the following if ans is 6 initially?

    (i) cout <<ans = 8 ; (ii) cout << ans == 8

Short answer type

1. What is a variable? List the two values associated with it.
2. In how many ways can a variable be declared in C++?
3. Explain the impact of type modifiers of C++ in variable declaration.
5. What is the role of the keyword 'const'?
6. Explain how prefix form of increment operation differs from postfix form.
8. Write down the operation performed by sizeof operator.
9. Explain the two methods of type conversions.
10. What would happen if `main()` is not present in a program?
11. Identify the errors in the following code segments:

    (a)
```
int main()
{ cout << "Enter two numbers"
cin >> num >> auto
float area = Length * breadth ; }
```
    (b)
```
#include <iostream>
using namespace std
void Main()
{ int a, b
cin <<a <<b
max=(a > b) a:b
cout>max
}
```
12. Find out the errors, if any, in the following + + statements:

    (i) `cout<< "a=" a;` (v) `cin >> "\n" >> y ;`

    (ii) `m=5,n=12;015` (vi) `cout >> \n "abc"`

(iii) `cout << "x" ; <<x;` (vii) `a = b + c`

(iv) `cin >> y` (viii) `break = x`

13. What is the role of relational operators? Distinguish between == and =.
14. Comments are useful to enhance readability and understandability of a program. Justify this statement with examples.

Long answer type

1. Explain the operators of C++ in detail.
2. Explain the different types of expressions in C++ and the methods of type conversions in detail.
3. Write the working of arithmetic assignment operator? Explain all arithmetic assignment operators with the help of examples.