1 Problem Solving and Programming

Takeaways

- Algorithms
- Iteration
- Design of efficient programs
- Flowcharts
- Generation of programming languages
- Types of errors
- Pseudocodes
- Modularization
- Testing and debugging approaches

1.1 INTRODUCTION

A *program* is a set of instructions that tells the computer how to solve a particular problem. Various program design tools like algorithms, pseudocdes and flowcharts are used to design the blueprint of the solution (or the program to be written). Computer programming goes a step further in problem-solving process. *Programming* means writing computer programs. While programming, the programmers take an algorithm and code the instructions in a particular programming language so that it can be executed by a computer. There are many programming languages available in the market now. The programmer can choose any language depending on their expertise and the problem domain.

The following sections will deal with different program design tools, knowledge of which is compulsory to develop programming skills.

1.2 ALGORITHMS

In computing, we focus on the type of problems categorically known as *algorithmic problems*, where the solutions are expressible in the form of algorithms. The term 'algorithm' was derived from the name of Mohammed al-Khwarizmi, a ninth-century Persian mathematician (Al-Khwarizmi → Algorism (in Latin) → Algorithm). The typical meaning of an algorithm is a formally defined procedure for performing some calculation. If a procedure is formally defined, then it must be implemented using some formal language, and such languages are known as *programming languages*. The algorithm gives the logic of the program, i.e., a step-by-step description of how to arrive at a solution.

In general terms, an algorithm provides a blueprint to writing a program to solve a particular problem. It is considered to be an effective procedure for solving a problem in a finite number of steps. That is, a well-defined algorithm always provides an answer, and is guaranteed to terminate.

Algorithms are mainly used to achieve *software reuse*. Once we have an idea or a blueprint of a solution, we can implement it in any language, such as C, C++, Java, and so on. In order to qualify as an algorithm, a sequence of instructions must possess the following characteristics:

- **Precision:** The instructions should be written in a precise manner.
- **Uniqueness:** The outputs of each step should be unambiguous, i.e., they should be unique and only depend on the input and the output of the preceding steps.

- **Finiteness:** Not even a single instruction must be repeated infinitely.
- **Effectiveness:** The algorithm should be designed in such a way that it should be the most effective among many different ways to solve a problem.
- **Input:** The algorithm must receive an input.
- **Output:** After the algorithm gets terminated, the desired result must be obtained.
- **Generality:** The algorithm can be applied to various sets of inputs.

1.3 BUILDING BLOCKS OF ALGORITHM (INSTRUCTIONS, STATE, CONTROL FLOW, FUNCTIONS)

An algorithm starts from an *initial state* with some input. The *instructions/statements* describe the processing that must be done on the input to produce the *final output* (the final state). Note that an *instruction* is a single operation which when executed converts one state to another.

In the course of processing, data is read from an input device and stored in computer's memory for further processing. The result of the processing is written to an output device.

The data is stored in the computer's memory in the form of variables or constants. The *state* of an algorithm is defined as its condition regarding current values or contents of the stored data.

An algorithm is a list of precise steps and the order of steps determines the functioning of the algorithm. The *flow of control* (or the control flow) of an algorithm can be specified as top-down or bottom-up approach. Thus, the flow of control specifies the order in which individual instructions of an algorithm are executed.

1.3.1 Subcharts/Subroutine/Predefined Process

A subroutine (or procedure or function or routine) is a sequence of instructions that performs a specific task. These instructions are packaged as a single unit and can be used (or invoked or called) wherever that particular task needs to be performed. After performing its defined task, the subroutine branches back (or *returns*) to the next instruction after the one that invoked it.

A subroutine may be designed to accept one or more data values (also known as parameters) from the calling code. It may also return a value to its caller. A subroutine can also be written in such a way that it calls itself repeatedly.

The subroutine symbol is used to write steps for procedures. These procedures can be called from anywhere in the code. This means that once the flowchart for a process is drawn, it can be referenced and used from anywhere in the code.

1.4 ALGORITHMIC PROBLEM SOLVING STEPS

As mentioned earlier, algorithms are solutions to problems. They are not solutions themselves. They just list specific instructions that need to be performed for getting the solution. In computer science, emphasis is laid on writing a good and effective algorithm and this emphasis makes computer science distinct from other disciplines. For example, computer science is distinct from theoretical mathematics because those practitioners are typically satisfied with just proving the existence of a solution to a problem but in computer science, the problem is not solved until the algorithm is used to implement the solution.

We will now discuss the sequence of steps one must typically follow for designing an effective algorithm.

1. Understanding the problem
2. Determining the capabilities of the computational device
3. Exact/approximate solution
4. Select the appropriate data structure
5. Algorithm design techniques
6. Methods of specifying an algorithm
7. Proving an algorithms correctness
8. Analysing the performance of an algorithm

Understanding the problem The problem given should be understood clearly and completely. It is compared with earlier problems that have already been solved to check if it is similar to them and a known algorithm exists. If the algorithm is available, it is used, otherwise a new one has to be developed.

Determining the capabilities of the computational device After understanding the problem, the capabilities of the computing device should be known. For this, the type of the architecture, speed and memory availability of the device are noted.

Exact/approximate solution The next step is to develop the algorithm. The algorithm must compute correct output for all possible and legitimate inputs. This solution can be an exact solution or an approximate solution. For example, you can only have an approximate solution in case of finding square root of number or finding the solutions of non-linear equations.

Select the appropriate data structure A *data type* is a well-defined collection of data with a well-defined set of operations on it. A *data structure* is basically a group of data elements that are put together under one name, and which defines a particular way of storing and organizing data in a computer so that it can be used efficiently. The elementary data structures are as follows.

- **List:** Allows fast access of data.
- **Sets:** Treats data as elements of a set. Allows application of operations such as intersection, union, and equivalence.
- **Dictionaries:** Allows data to be stored as a key-value pair.

Algorithm design techniques Developing an algorithm is an art which may never be fully automated. By mastering the design techniques, it will become easier for you to develop new and useful algorithms. Examples of algorithm design techniques include dynamic programming.

Methods of specifying an algorithm An algorithm is just a sequence of steps or instructions that can be used to implement a solution. After writing the algorithm, it is specified either using a natural language or with the help of pseudocode and flowcharts. We will read about them in the next section.

Proving algorithms correctness Writing an algorithm is not just enough. You need to prove that it computes solutions for all the possible valid inputs. This process is often referred to as algorithm validation. Algorithm validation ensures that the algorithm will work correctly irrespective of the programming language in which it will be implemented.

Analysing the performance of algorithms When an algorithm is executed, it uses the computer's resources like the Central Processing Unit (CPU) to perform its operation and to hold the program and data respectively. An algorithm is analysed to measure its performance in terms of CPU time and memory space required to execute that algorithm. This is a challenging task and is often used to compare different algorithms for a particular problem. The result of the comparison helps us to choose the best solution from all possible solutions. Analysis of the algorithm also helps us to determine whether the algorithm will be able to meet any efficiency constraint that exits or not.

1.5 SIMPLE STRATEGIES AND NOTATIONS FOR DEVELOPING ALGORITHMS

An algorithm is a *step–by–step procedure* for solving a task or problem. However, these steps must be ordered, unambiguous and finite in number. Basically, an algorithm is nothing but English-like representation of logic which is used to solve the problem.

For accomplishing a particular task, different algorithms can be written. The different algorithms differ in their requirements of CPU time and memory space. The programmer selects the best suited algorithm for the given task to be solved.

Various strategies and notations used for developing and designing algorithms are discussed in the following sections.

1.5.1 Control Structures Used in Algorithms

An algorithm has a finite number of steps and some steps may involve decision making and repetition. Broadly speaking, an algorithm may employ three control structures, namely, sequence, decision, and repetition.

Sequence Sequence means that each step of the algorithm is executed in the specified order. An algorithm to add two numbers is given as follows. This algorithm performs the steps in a purely sequential order.

Example 1.1

Algorithm to add two numbers

```
Step 1 : Start
Step 2 : Input first number as A
Step 3 : Input second number as B
Step 4 : Set Sum = A + B
Step 5 : Print Sum
Step 6 : End
```

Decision Decision statements are used when the outcome of the process depends on some condition. For example, if x = y, then print "EQUAL". Hence, the general form of the if construct can be given as follows:

```
IF condition then process
```

A condition in this context is any statement that may evaluate either to a true value or a false value. In the preceding example, the variable x can either be equal or not equal to y. However, it cannot be both true and false. If the condition is true then the process is executed.

A decision statement can also be stated in the following manner:

```
IF condition
   then process1
ELSE process2
```

This form is commonly known as the if-else construct. Here, if the condition is true then process1 is executed, else process2 is executed. An algorithm to check the equality of two numbers is shown below.

Example 1.2

Algorithm to test the quality of two numbers

```
Step 1 : Start
Step 2 : Input first number as A
Step 3 : Input second number as B
Step 4 : IF A = B
          Print "Equal"
         ELSE
          Print "Not equal" [END of IF]
Step 5 : End
```

Let us look at the following two simple algorithms to find the greatest among three numbers.

Example 1.3

Algorithm to find the greatest of three numbers

```
Step 1: Start
Step 2: Read the three numbers A,B,C
Step 3: Compare A and B. If A is greater
perform step 4 else perform step 5.
Step 4: Compare A and C. If A is greater,
        output "A is greatest" else output
        "C is greatest"
Step 5: Compare B and C. If B is greater,
        output "B is greatest" else output
        "C is greatest"
Step 6: End
```

Example 1.4

Algorithm to find the greatest of three numbers using an additional variable MAX

```
Step 1: Start
Step 2: Read the three numbers A,B,C
Step 3: Compare A and B. If A is greater,
store Ain MAX, else store B in MAX
Step 4: Compare MAX and C. If MAX is
        greater, output "MAX is greater"
        else output "C is greater"
Step 5: End
```

Both the algorithms given in Examples 1.3 and 1.4 accomplish same goal, but in different ways. The programmer selects the algorithm based on the advantages and disadvantages of each algorithm. For example, the first algorithm has more number of comparisons, whereas in the second algorithm an additional variable MAX is used to do the comparison.

Iteration or repetition which involves executing one or more steps for a number of times, can be implemented using constructs such as the while loops and for loops. These loops execute one or more steps until some condition is true.

Example 1.5

Algorithm that prints the first 10 natural numbers

```
Step 1: Start
Step 2: [initialize] Set I = 1, N = 10
Step 3: Repeat Steps 3 and 4 while I <= N
Step 4: Print I
Step 5: Set I = I + 1 [END OF LOOP]
Step 6: End
```

Example 1.6

Design an algorithm for adding the test scores given as: 26, 49, 98, 87, 62, 75.

```
Step 1: Start
Step 2: sum = 0
Step 3: Get a value
Step 4: sum = sum + value
Step 5: If next value is present, go to
step 3. Otherwise, go to step 6
Step 6: Print the sum
Step 7: End
```

Examples 1.5 and 1.6 demonstrate the application of repetition and iteration logic in an algorithm.

Some more examples to depict the concept of control structures in algorithms are given as follows.

Example 1.7

Write an algorithm for interchanging/swapping two values.

```
Step 1: Start
Step 2: Input first number as A
```

```
Step 3: Input second number as B
Step 4: Set temp = A
Step 5: Set A = B
Step 6: Set B = temp
Step 7: Print A, B
Step 8: End
```

Example 1.8

Write an algorithm to find the larger of two numbers.

```
Step 1: Start
Step 2: Input first number as A
Step 3: Input second number as B
Step 4: IF A > B
            Print A
        ELSE IF A < B
            Print B
        ELSE
            Print "The numbers are equal"
        [END OF IF]
Step 5: End
```

Example 1.9

Write an algorithm to find whether a number is even or odd.

```
Step 1: Start
Step 2: Input number as A
Step 3: IF A % 2 = 0
          Print "Even"
        ELSE
          Print "Odd"
        [END OF IF]
Step 4: End
```

Example 1.10

Write an algorithm to print the grade obtained by a student using the following rules:

Marks	Grade
Above 75	O
60-75	A
50-60	B
40-50	C
Less than 40	D

```
Step 1: Start
Step 2: Enter the marks obtained as M
Step 3: IF M > 75
          Print "O"
        ELSE IF M >= 60 and M < 75
          Print "A"
        ELSE IF M >= 50 and M < 60
          Print "B"
        ELSE IF M >= 40 and M < 50
          Print "C"
        ELSE
          Print "D"
        [END OF IF]
Step 6: End
```

Example 1.11

Write an algorithm to find the sum of first N natural numbers.

```
Step 1: Start
Step 2: Input N
Step 3: Set I = 1, sum = 0
Step 4: Repeat Steps 4 and 5 while I <= N
Step 5: Set sum = sum + I
Step 6: Set I = I + 1
        [END OF LOOP]
Step 7: Print sum
Step 8: End
```

Recursion It is a technique of solving a problem by breaking it down into smaller and smaller sub-problems until you get to a small enough problem that it can be easily solved. Usually, recursion involves a function calling itself until a specified condition is met. Since a recursive function repeatedly calls itself, it makes use of the system stack to temporarily store the return address and local variables of the calling function. Every recursive solution has two major cases. They are:

- ***Base case***, in which the problem is simple enough to be solved directly without making any further calls to the same function.
- ***Recursive case***, in which first the problem at hand is divided into simpler sub-parts. Second the function calls itself but with sub-parts of the problem obtained in the first step. Third, the result is obtained by combining the solutions of simpler sub-parts.

Therefore, recursion is defining large and complex problems in terms of smaller and more easily solvable problems. In recursive functions, a complex problem is defined in terms of simpler problems and the simplest problem is given explicitly.

(For a detailed study on Recursion and Iteration, refer to Chapter 4)

Example 1.12

Write a recursive algorithm to find the factorial of a number.

```
Step 1: Start
Step 2: Input number as n
```

```
Step 3: Call factorial(n)
Step 4: End

factorial(n)
Step 1: Set f = 1
Step 2: IF n==1 then return 1
        ELSE
         Set f=n*factorial(n-1)
Step 3: Print f
```

1.5.2 Flowcharts

A flowchart is a graphical or symbolic representation of a process. It is basically used to design and document virtually complex processes to help the viewers to visualize the logic of the process, so that they can gain a better understanding of the process and find flaws, bottlenecks, and other less obvious features within it.

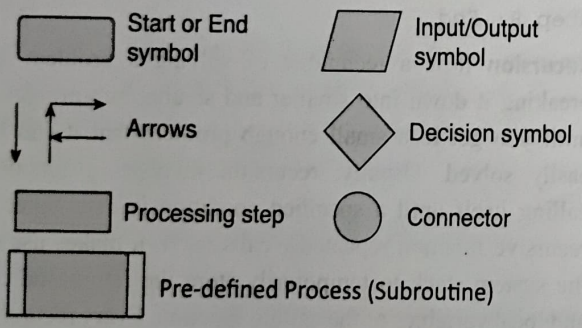


Figure 1.1 Symbols used in a Flowchart

When designing a flowchart, each step in the process is depicted by a different symbol and is associated with a short description. The symbols in the flowchart (refer Figure 1.1) are linked together with arrows to show the flow of logic in the process.

The symbols used in a flowchart include the following:

- *Start and end symbols* are also known as the terminal symbols and are represented as circles, ovals, or rounded rectangles. Terminal symbols are always the first and the last symbols in a flowchart.
- *Arrows* depict the flow of control of the program. They illustrate the exact sequence in which the instructions are executed.
- *Generic processing step,* also called as an activity, is represented using a rectangle. Activities include instructions such as *add a to b* or *save the result.* Therefore, a processing symbol represents arithmetic and data movement instructions. When more than one process has to be executed simultaneously, they can be placed in the same processing box. However, their execution will be carried out in the order of their appearance.
- *Input/Output symbols* are represented using a parallelogram and are used to get inputs from the users or display the results to them.
- A *conditional or decision symbol* is represented using a diamond. It is basically used to depict a Yes/No question or a True/False test. The two symbols coming out of it, one from the bottom point and the other from the right point, corresponds to Yes or True, and No or False, respectively. The arrows should always be labelled. A decision symbol in a flowchart can have more than two arrows, which indicates that a complex decision is being taken.
- *Labelled connectors* are represented by an identifying label inside a circle and are used in complex or multi-sheet diagrams to substitute for arrows. For each label, the 'outflow' connector must have one or more 'inflow' connectors. A pair of identically labelled connectors is used to indicate a continued flow when the use of lines becomes confusing.
- A *pre-defined process symbol* is a marker for another process step or series of process flow steps that are formally defined elsewhere. This shape commonly depicts sub-processes (or subroutines in programming flowcharts).

Significance of Flowcharts

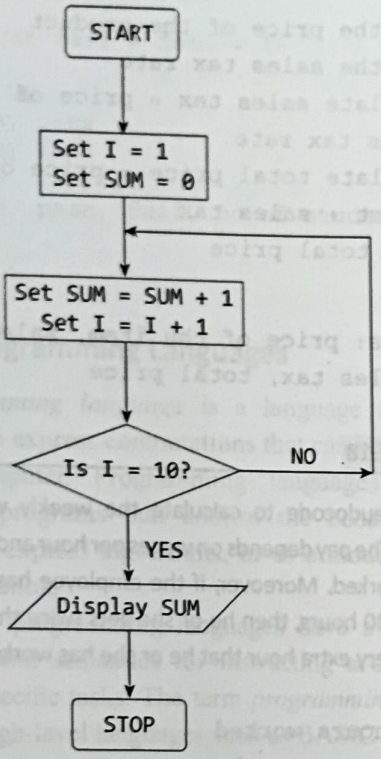
A flowchart is a diagrammatic representation that illustrates the sequence of steps that must be performed to solve a problem. It is usually drawn in the early stages of formulating computer solutions. It facilitates communication between programmers and users. Once a flowchart is drawn, programmers can make users understand the solution easily and clearly.

Flowcharts are very important in the programming of a problem as they help the programmers to understand the logic of complicated and lengthy problems. Once a flowchart is drawn, it becomes easy for the programmers to write the program in any high-level language. Hence, the flowchart has become a necessity for better documentation of complex programs.

A flowchart follows the top-down approach in solving problems. Some examples are given as follows.

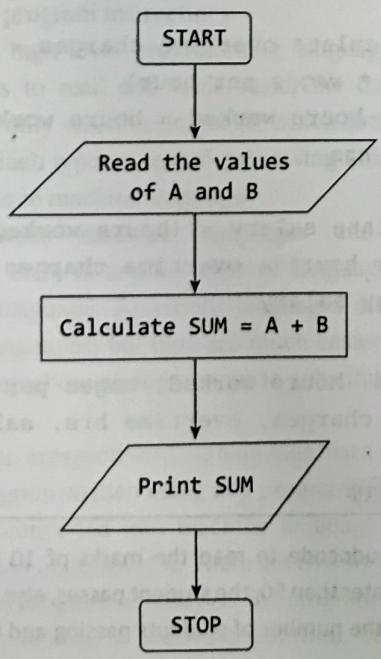
Example 1.13

Draw a flowchart to calculate the sum of the first 10 natural numbers.



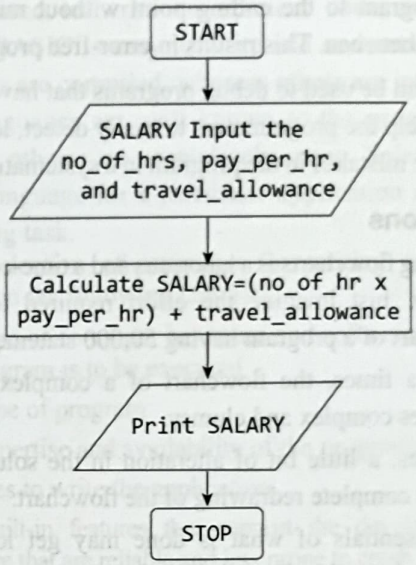
Example 1.14

Draw a flowchart to add two numbers.



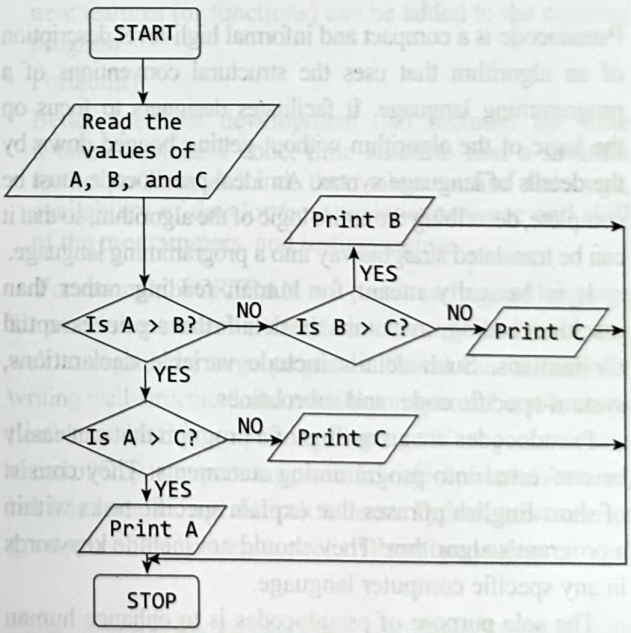
Example 1.15

Draw a flowchart to calculate the salary of a daily wager.



Example 1.16

Draw a flowchart to determine the largest of three numbers.



Advantages

- They are very good communication tools to explain the logic of a system to all concerned. They help to analyse the problem in a more effective manner.
- They are also used for program documentation. They are even more helpful in the case of complex programs.

- They act as a guide or blueprint for the programmers to code the solution in any programming language. They direct the programmers to go from the starting point of the program to the ending point without missing any step in between. This results in error-free programs.
- They can be used to debug programs that have error(s). They help the programmers to easily detect, locate, and remove mistakes in the program in a systematic manner.

Limitations

- Drawing flowcharts is a laborious and a time-consuming activity. Just imagine the effort required to draw a flowchart of a program having 50,000 statements in it!
- Many a times, the flowchart of a complex program becomes complex and clumsy.
- At times, a little bit of alteration in the solution may require complete redrawing of the flowchart.
- The essentials of what is done may get lost in the technical details of how it is done.
- There are no well-defined standards that limit the details that must be incorporated into a flowchart.

1.5.3 Pseudocodes

Pseudocode is a compact and informal high-level description of an algorithm that uses the structural conventions of a programming language. It facilitates designers to focus on the logic of the algorithm without getting bogged down by the details of language syntax. An ideal pseudocode must be complete, describing the entire logic of the algorithm, so that it can be translated straightaway into a programming language.

It is basically meant for human reading rather than machine reading, so it omits the details that are not essential for humans. Such details include variable declarations, system-specific code, and subroutines.

Pseudocodes are an outline of a program that can easily be converted into programming statements. They consist of short English phrases that explain specific tasks within a program's algorithm. They should not include keywords in any specific computer language.

The sole purpose of pseudocodes is to enhance human understandability of the solution. They are commonly used in textbooks and scientific publications for documenting algorithms, and for sketching out the program structure before the actual coding is done. This helps even non-programmers to understand the logic of the designed solution. There are no standards defined for writing a pseudocode, because a pseudocode is not an executable program. Flowcharts can be considered as graphical alternatives to pseudocodes, but require more space on paper.

Example 1.17

Write a pseudocode for calculating the price of a product after adding the sales tax to its original price.

```
1. Start
2. Read the price of the product
3. Read the sales tax rate
4. Calculate sales tax = price of the item
x * sales tax rate
5. Calculate total price = price of the
   product + sales tax
6. Print total price
7. End

Variables: price of the item, sales tax
rate, sales tax, total price
```

Example 1.18

Write a pseudocode to calculate the weekly wages of an employee. The pay depends on wages per hour and the number of hours worked. Moreover, if the employee has worked for more than 30 hours, then he or she gets twice the wages per hour, for every extra hour that he or she has worked.

```
1. Start
2. Read hours worked
3. Read wages per hour
4. Set overtime charges to 0
5. Set overtime hrs to 0
6. IF hours worked > 30 then
   a. Calculate overtime hrs = hours worked
- 30
   b. Calculate overtime charges = overtime
hrs * (2 * wages per hour)
   c. Set hours worked = hours worked -
overtime hrs
   ENDIF
7. Calculate salary = (hours worked x
wages per hour) + overtime charges
8. Display salary
9. End

Variables: hours worked, wages per hour,
overtime charges, overtime hrs, salary
```

Example 1.19

Write a pseudocode to read the marks of 10 students. If marks is greater than 50, the student passes, else the student fails. Count the number of students passing and failing.

```
1. Start
2. Set pass to 0
```

```
3. Set fail to 0
4. Set no of students to 1
5. WHILE no of students < 10
   a. input the marks
   b. IF marks >= 50 then
         Set pass = pass + 1
   ELSE
         Set fail = fail + 1
   ENDIF
   ENDWHILE
6. Display pass
7. Display fail
8. End
Variables: pass, fail, no of students,
marks
```

1.5.4 Programming Languages

A *programming language* is a language specifically designed to express computations that can be performed by a computer. Programming languages are used to create programs that control the behaviour of a system, to express algorithms, or as a mode of human communication.

Usually, programming languages have a vocabulary of syntax and semantics for instructing a computer to perform specific tasks. The term *programming language* refers to high-level languages such as BASIC (Beginners' All-purpose Symbolic Instruction Code), C, C++, COBOL (Common Business Oriented Language), FORTRAN (Formula Translator), Python, Ada, and Pascal, to name a few. Each of these languages has a unique set of keywords (words that it understands) and a special syntax for organizing program instructions.

Though high-level programming languages are easy for humans to read and understand, the computer can understand only machine language, which consists of only numbers. Each type of central processing unit (CPU) has its own unique machine language.

In between machine languages and high-level languages, there is another type of language known as assembly language. Assembly languages are similar to machine languages, but they are much easier to program because they allow a programmer to substitute names for numbers.

However, irrespective of the language that a programmer uses, a program written using any programming language has to be converted into machine language so that the computer can understand it. There are two ways to do this: *compile* the program or *interpret* the program.

When planning a software solution, the software development team often faces a common question—which programming language to use? Many programming languages are available today and each one has its own strengths and weaknesses. Python can be used to write an efficient code, whereas a code in BASIC is easy to write and understand; some languages are compiled, whereas others are interpreted; some languages are well known to the programmers, whereas others are completely new. Selecting the perfect language for a particular application at hand is a daunting task.

The selection of language for writing a program depends on the following factors:

- The type of computer hardware and software on which the program is to be executed
- The type of program
- The expertise and availability of the programmers
- Features to write the application
- The built-in features that support the development of software that are reliable and less prone to crash
- Lower development and maintenance costs
- Stability and capability to support even more than the expected simultaneous users
- Elasticity of a language that implies the ease with which new features (or functions) can be added to the existing program
- Portability
- Better speed of development that includes the time it takes to write a code, time taken to find a solution to the problem at hand, time taken to find the bugs, availability of development tools, experience and skill of the programmers, and testing regime

For example, FORTRAN is a particularly good language for processing numerical data, but it does not lend itself very well to organizing large programs. Pascal can be used for writing well-structured and readable programs, but it is not as flexible as the C programming language. C++ goes one step ahead of C by incorporating powerful object oriented features, but it is complex and difficult to learn. Python, however is a good mix of the best features of all these languages.

1.6 GENERATION OF PROGRAMMING LANGUAGES

We now know that programming languages are the primary tools for creating software. As of now, hundreds of programming languages exist in the market, some more used than others, and each claiming to be the best. However, back in the 1940s when computers were being developed there was just one language—the machine language.

The concept of generations of programming languages (also known as levels) is closely connected to the advances in technology that brought about computer generations. The four generations of programming languages are:

- Machine language
- Assembly language
- High-level language (also known as third generation language or 3GL)
- Very high-level language (also known as fourth generation language or 4GL)

1.6.1 First Generation: Machine Language

Machine language was used to program the first stored program on computer systems. This is the lowest level of programming language. The machine language is the only language that the computer understands. All the commands and data values are expressed using 1 and 0s, corresponding to the 'on' and 'off' electrical states in a computer.

In the 1950s each computer had its own native language, and programmers had primitive systems for combining numbers to represent instructions such as *add* and *subtract*. Although there were similarities between each of the machine languages, a computer could not understand programs written in another machine language (Figure 1.2).

MACHINE LANGUAGE

This is an example of a machine language program that will add two numbers and find their average. It is in hexadecimal notation instead of binary notation because this is how the computer presented the code to the programmer.

```
                                        D000000A    D000
                                        D000000F    D009
                                        D000000B    D009
                                                    D009
                                                    D009
                                                    D0Θ0
FF55  CF    FF54  CF    FF53  CF  C1    DOD0
      FF24  CF    FF27  CF    D2  C7    D00C
                                        D0E4
                                        Dd0D
                                        Dd3D
```

Figure 1.2 A machine language program

In machine language, all instructions, memory locations, numbers, and characters are represented in strings of 1s and 0s. Although machine-language programs are typically displayed with the binary numbers represented in octal (base 8) or hexadecimal (base 16), these programs are not easy for humans to read, write, or debug.

The main advantage of machine language is that the code can run very fast and efficiently, since it is directly executed by the CPU.

However, on the downside, the machine language is difficult to learn and is far more difficult to edit if errors occur. Moreover, if you want to add some instructions into memory at some location, then all the instructions after the insertion point would have to be moved down to make room in memory to accommodate the new instructions.

Last but not the least, the code written in machine language is not portable across systems and to transfer the code to a different computer it needs to be completely rewritten since the machine language for one computer could be significantly different from another computer. Architectural considerations made portability a tough issue to resolve.

1.6.2 Second Generation: Assembly Language

The second generation of programming language includes the assembly language. Assembly languages are symbolic programming languages that use symbolic notation to represent machine-language instructions. These languages are closely connected to machine language and the internal architecture of the computer system on which they are used. Since they are close to the machine, assembly language is also called low-level language. Nearly all computer systems have an assembly language available for use.

Assembly language developed in the mid 1950s was a great leap forward. It used symbolic codes also known as *mnemonic codes* that are easy-to-remember abbreviations, rather than numbers. Examples of these codes include ADD for add, CMP for compare, MUL for multiply, etc.

Assembly language programs consist of a series of individual statements or instructions that instruct the computer what to do. Basically, an assembly language statement consists of a *label*, an *operation code*, and one or more *operands*.

Labels are used to identify and reference instructions in the program. The operation code (opcode) is a mnemonic that specifies the operation that has to be performed such

as *move, add, subtract,* or *compare*. The operand specifies the register or the location in main memory where the data to be processed is located.

However, like the machine language, the statement or instruction in the assembly language will vary from machine to another because the language is directly related to the internal architecture of the computer and is not designed to be machine independent. This makes the code written in assembly language less portable as the code written for one machine will not run on machines from a different or sometimes even the same manufacturer.

No doubt, the code written in assembly language will be very efficient in terms of execution time and main memory usage as the language is also close to the computer.

Programs written in assembly language need a *translator* often known as *assembler* to convert them into machine language. This is because the computer will understand only the language of 1s and 0s and will not understand mnemonics like ADD and SUB.

The following instructions are a part of assembly language code to illustrate addition of two numbers:

```
MOV AX,4      Stores value 4 in the AX
              register of CPU
MOV BX,6      Stores value 6 in the BX
              register of CPU
ADD AX,BX     Adds the contents of AX and BX
              registers. Stores the result in
              AX register
```

Although assembly languages are much better to program as compared to the machine language, they still require the programmer to think on the machine's level. Even today, some programmers still use assembly language to write parts of applications where speed of execution is critical, such as video games but most programmers today have switched to third or fourth generation programming languages.

1.6.3 Third Generation Programming Languages

A third generation programming language (3GL) is a refinement of the second-generation programming language. The 2GL languages brought logical structure to software. The third generation was introduced to make the languages more programmer friendly.

Third Generation Programming Languages spurred the great increase in data processing that occurred in the 1960s and 1970s. In these languages, the program statements are not closely related to the internal architecture of the computer and is therefore often referred to as high-level languages.

Generally, a statement written in a high-level programming language will expand into several machine language instructions. This is in contrast to assembly languages, where one statement would generate one machine language instruction. Third Generation Programming Languages made programming easier, efficient, and less prone to errors.

High-level languages fall somewhere between natural languages and machine languages. Third Generation Programming Languages include languages such as FORTRAN (FORmula TRANslator) and COBOL (COmmon Business Oriented Language) that made it possible for scientists and business people to write programs using familiar terms instead of obscure machine instructions.

The first widespread use of high-level languages in the early 1960s changed programming into something quite different from what it had been. Programs were written in statements like English language statements, making them more convenient to use and giving the programmer more time to address a client's problems.

Although 3GLs relieve the programmer of demanding details, they do not provide the flexibility available in low-level languages. However, a few high-level languages like C and FORTRAN combine some of the flexibility of assembly language with the power of high-level languages, but these languages are not well suited to an amateur programmer.

While some high-level languages were designed to serve a specific purpose (such as controlling industrial robots or creating graphics), other languages were flexible and considered to be general-purpose languages. Most of the programmers preferred to use general-purpose high-level languages like BASIC (Beginners' All-purpose Symbolic Instruction Code), FORTRAN, PASCAL, COBOL, C++, or Java to write the code for their applications.

Again, a *translator* is needed to translate the instructions written in high-level language into computer-executable machine language. Such translators are commonly known as interpreters and compilers. Each high-level language has many compilers.

For example, the machine language generated by one computer's C compiler is not the same as the machine language of some other computer. Therefore, it is necessary to have a C compiler for each type of computer on which the C program has to be executed.

Third generation programming languages have made it easier to write and debug programs, which gives programmers more time to think about its overall logic.

The programs written in such languages are portable between machines. For example, a program written in standard C can be compiled and executed on any computer that has a standard C compiler.

1.6.4 Fourth Generation: Very High-Level Languages

With each generation, programming languages started becoming easier to use and more like natural languages. However, fourth generation programming languages (4GLs) are a little different from their its prior generation because they are basically non-procedural. When writing code using a procedural language, the programmer has to tell the computer how a task is done—add this, compare that, do this if the condition is true, and so on, in a very specific step-by-step manner. In striking contrast, while using a non-procedural language the programmers define only what they want the computer to do, without supplying all the details of how it has to be done.

There is no standard rule that defines what a 4GL is but certain characteristics of such languages include:

- the code comprising instructions are written in English-like sentences;
- they are non-procedural, so users concentrate on 'what' instead of the 'how' aspect of the task;
- the code is easier to maintain;
- the code enhances the productivity of the programmers as they have to type fewer lines of code to get something done. It is said that a programmer becomes 10 times more productive when he writes the code using a 4GL than using a 3GL.

A typical example of a 4GL is the *query language* that allows a user to request information from a database with precisely worded English-like sentences. A query language is used as a database user interface and hides the specific details of the database from the user. For example, when working with structured query language (SQL), the programmer just needs to remember a few rules of *syntax* and *logic*, and it is easier to learn than COBOL or C.

Let us take an example in which a report has to be generated that displays the total number of students enrolled in each class and in each semester. Using a 4GL, the request would look similar to one that follows:

```
TABLE FILE ENROLLMENT

SUM STUDENTS BY SEMESTER BY CLASS
```

So we see that a 4GL is much simpler to learn and work with. The same code if written in C language or any other 3GL would require multiple lines of code to do the same task.

Fourth generation programming languages are still evolving, which makes it difficult to define or standardize them. The only downside of a 4GL is that it does not make efficient use of the machine's resources. However, the benefit of executing a program fast and easily, far outweighs the extra costs of running it.

1.6.5 Fifth Generation Programming Languages

Fifth generation programming languages (5GLs) are centred on solving problems using constraints given to the program, rather than using an algorithm written by a programmer. Most constraint-based and logic programming languages and some declarative languages form a part of the fifth-generation languages. Fifth generation programming languages are widely used in artificial intelligence research. Typical examples of 5GLs include Prolog, OPS5, and Mercury.

Another aspect of a 5GL is that it contains visual tools to help develop a program. A good example of a fifth generation language is Visual Basic.

So taking a forward leap than the 4GLs, 5GLs are designed to make the computer solve a given problem without the programmer. While working with a 4GL, the programmer had to write specific code to do a work but with 5GL, the programmer only needs to worry about what problems need to be solved and what conditions need to be met, without worrying about how to implement a routine or algorithm to solve them.

Generally, 5GLs were built upon Lisp, many originating on the Lisp machine, such as ICAD. Then, there are many frame languages such as KL-ONE.

In the 1990s, 5GLs were considered to be the wave of the future, and some predicted that they would replace all other languages for system development (except the low-level languages). In 1982 to 1993 Japan had put much research and money into their fifth generation computer systems project, hoping to design a massive computer network of machines using these tools. But when larger programs were built, the flaws of the approach became more apparent. Researchers began to observe that starting from a set of constraints for defining a particular problem, then deriving an efficient algorithm to solve the problem is a very difficult task. All these things could not be automated and still requires the insight of a programmer.

However, today the fifth-generation languages are back as a possible level of computer language. Software vendors across the globe currently claim that their software meets the visual 'programming' requirements of the 5GL concept.

1.7 MODULARIZATION

A structured programming language like C employs a top-down approach in which the overall program structure is broken down into separate modules. Each module is coded separately and once it is written and tested individually, it is then integrated with other modules to form the overall program structure. This allows the code to be loaded into memory more efficiently and also be reused in other programs.

Structured programming is therefore based on modularization which groups related statements (modules) together. Modularization makes it easier to write, debug, and understand programs.

Ideally, modules should not be longer than a page. It is always easy to understand a series of ten single-page modules than a single ten-page program.

For some large and complex programs, the overall program structure may further require the need to break the modules into smaller pieces. This process continues until an individual piece of code can be written easily.

Almost any language can use structured programming techniques to avoid common pitfalls of unstructured languages. Unstructured programs depend on the programmer's skills to avoid structural problems and are therefore poorly organized. Most modern procedural languages support the concept of structured programming. Even the object-oriented programming can be thought of as a type of structured programming, as it uses the techniques of structured programming for program flow, and adds more structure to model the data.

In structured programming, the program flow follows a simple sequence and usually avoids the use of `Goto` statements. Besides sequential flow, structured programming also supports selection and repetition. Selection allows for choosing any one of a number of statements to execute, based on the outcome of a condition. Selection statements contain keywords like "if," "then," "endif," or "switch" that help to identify the order as logically executable.

In repetition, a selected statement remains active until the program reaches a point where there is a need for some other action to take place. It includes keywords like "repeat," "for," or "do...until." Essentially, repetition instructs the program how long to continue the function before requesting further instructions.

Advantages

- The goal of structured programming is to write correct programs that are easy to understand and modify.
- Modules enhance programmers' productivity by allowing them to look at the big picture first and then focus on details later.
- Using modules, many programmers can work on a single, large program, with each programmer working on a different module.
- A structured program takes less time to be written than other programs. Modules or procedures written for one program can be reused in other programs also.
- A structured program is easy to debug because each procedure is specialized to perform just one task so every procedure can be checked individually for the presence of any error. On the other hand, unstructured programs consist of a sequence of instructions that are not grouped for specific tasks. Their logic is cluttered with details and therefore difficult to follow.
- Individual procedures are easy to change as well as understand. In a structured program every procedure has meaningful names and has clear documentation to identify the task performed by it. Moreover, a correctly written structured program is self-documenting and can be easily understood by another programmer.

Example 1.20

Imagine that your institute wants to create a program to manage a list of names and addresses of students.

For this, you would break down the program into the following modules:

1. Enter new entries
2. Modify existing entries
3. Sort entries
4. Print the list

Now each of the above modules can be further broken down into smaller modules. For example, *Enter new entries* module can be subdivided into modules like

a. Prompt the user to enter new data

b. Read the existing list from the disk

c. Add the name and address to the existing list

d. Save the updated list to the disk

Similarly, *Modify existing entries* can be further divided into modules like:

a. Read the existing list from the disk

b. Modify one or more entries

c. Save the updated list to the disk

Observe that two sub-modules: *Read the existing list from disk* and *Save the updated list to disk* are common to both the modules. So once these sub-modules are written they can be used by both the modules which require the same task to be performed. Structured programming method results in a hierarchical or layered program structure, which is depicted in Figure 1.3.

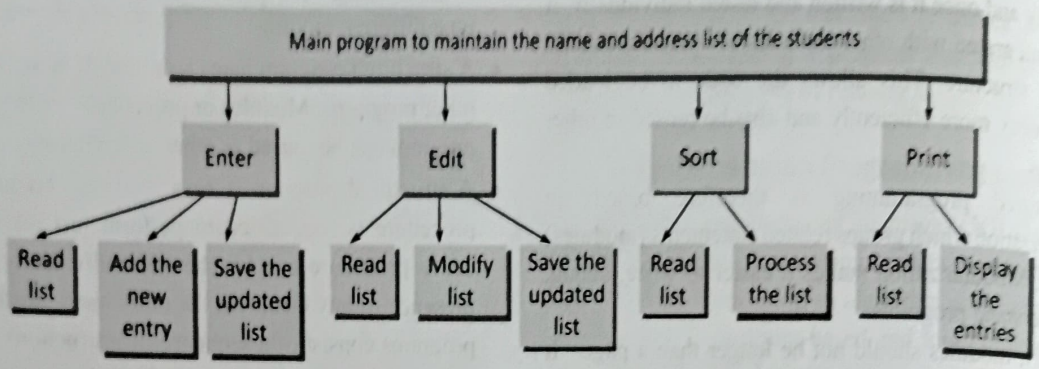


Figure 1.3 Layered program structure

1.8 DESIGN AND IMPLEMENTATION OF EFFICIENT PROGRAMS

The design and development of correct, efficient, and maintainable programs depends on the approach adopted by the programmer to perform various activities that need to be performed during the development process. The entire program or software (collection of programs) development process is divided into a number of phases where each phase performs a well-defined task. Moreover, the output of one phase provides the input for its subsequent phase.

The phases in software development process (as shown in Figure 1.4) can be summarized as follows:

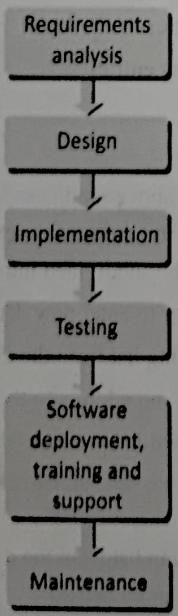


Figure 1.4 Phases in software development life cycle

1.8.1 Requirements Analysis

In this phase, users' expectations are gathered to know why the program/software has to be built. Then all the gathered requirements are analysed to pen down the scope or the objective of the overall software product. The last activity in this phase includes documenting every identified requirement of the users in order to avoid any doubts or uncertainty regarding the functionality of the programs. The functionality, capability, performance, availability of hardware and software components are all analysed in this phase.

1.8.2 Design

The requirements documented in the previous phase acts as an input to the design phase. In the design phase, a plan of actions is made before the actual development process could start. This plan will be followed throughout the development process. Moreover, in the design phase the core structure of the software/program is broken down into modules. The solution of the program is then specified for each module in the form of algorithms, flowcharts, or pseudocodes. The design phase, therefore, specifies how the program/software will be built.

1.8.3 Implementation

In this phase, the designed algorithms are converted into program code using any of the high level languages. The particular choice of language will depend on the type of program like whether it is a system or an application program. While C is preferred for writing system programs, Visual Basic might be preferred for writing an

application program. The program codes are tested by the programmer to ensure their correctness.

This phase is also called construction or code generation phase as the code of the software is generated in this phase. While constructing the code, the development team checks whether the software is compatible with the available hardware and other software components that were mentioned in the Requirements Specification Document created in the first phase.

1.8.4 Testing

In this phase, all the modules are tested together to ensure that the overall system works well as a whole product. Although individual pieces of codes are already tested by the programmers in the implementation phase, there is always a chance for bugs to creep in the program when the individual modules are integrated to form the overall program structure. In this phase, the software is tested using a large number of varied inputs also known as test data to ensure that the software is working as expected by the users' requirements that were identified in the requirements analysis phase.

1.8.5 Software Deployment, Training, and Support

After the code is tested and the software or the program has been approved by the users, it is then installed or deployed in the production environment. Software Training and Support is a crucial phase which is often ignored by most of the developers. Program designers and developers spend a lot of time to create software but if nobody in an organization knows how to use it or fix up certain problems, then no one would like to use it. Moreover, people are often resistant to change and avoid venturing into an unfamiliar area, so as a part of the deployment phase, it has become very crucial to have training classes for the users of the software.

1.8.6 Maintenance

Maintenance and enhancements are ongoing activities which are done to cope with newly discovered problems or new requirements. Such activities may take a long time to complete as the requirement may call for addition of new code that does not fit the original design or an extra piece of code required to fix an unforeseen problem. As a general rule, if the cost of the maintenance phase exceeds 25% of the prior-phases cost then it clearly indicates that the overall quality of at least one prior phase is poor. In such cases, it is better to re-build the software (or some modules) before maintenance cost is out of control.

1.9 TYPES OF ERRORS

While writing programs, very often we get errors in our program. These errors if not removed will either give erroneous output or will not let the compiler to compile the program. These errors are broadly classified under four groups as shown in Figure 1.5

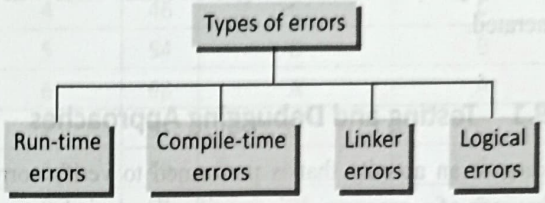


Figure 1.5 Types of Errors

Run-time Errors As the name suggests, run-time errors occur when the program is being run executed. Such errors occur when the program performs some illegal operations like

- Dividing a number by zero
- Opening a file that already exists
- Lack of free memory space
- Finding square or logarithm of negative numbers

Run-time errors may terminate program execution, so the code must be written in such a way that it handles all sorts of unexpected errors rather terminating it unexpectedly. This ability to continue operation of a program despite of run-time errors is called robustness.

Compile-time Errors Again as the name implies, compile-errors occur at the time of compilation of the program. Such errors can be further classified as follows:

Syntax Errors Syntax error is generated when rules of `C` programming language are violated. For example, if we write `int a:` then a syntax error will occur since the correct statement should be `int a;`

Semantic Errors Semantic errors are those errors which may comply with rules of the programming language but are not meaningful to the compiler. For example, if we write, `a * b = c;` it does not seem correct. Rather, if written like `c = a * b` would have been more meaningful.

Logical Errors Logical errors are errors in the program code that result in unexpected and undesirable output which is obviously not correct. Such errors are not detected by the compiler, and programmers must check their code line by line or use a debugger to locate and rectify the errors. Logical errors occur due to incorrect statements. For example, if you meant to perform `c = a + b;` and by

mistake you typed `c = a * b;` then though this statement is syntactically correct, it is logically wrong.

Linker Errors These errors occur when the linker is not able to find the function definition for a given prototype. For example, if you have defined a function `display_data()` but while calling if you mistakenly write `displaydata()` then again a linker error will be generated.

1.9.1 Testing and Debugging Approaches

Testing is an activity that is performed to verify correct behaviour of a program. It is specifically carried out with an intent to find errors. Ideally testing should be conducted at all stages of program development. However, in the implementation stage, three types of tests can be conducted:

Unit Tests Unit testing is applied only on a single unit or module to ensure whether it exhibits the expected behaviour.

Integration Tests These tests are a logical extension of unit tests. In this test, two units that have already been tested are combined into a component and the interface between them is tested. The guiding principle is to test combinations of pieces and then gradually expanding the component to include other modules as well. This process is repeated until all the modules are tested together. The main focus of integration testing is to identify errors that occur when the units are combined.

System Tests System testing checks the entire system. For example, if our program code consists of three modules then each of the module is tested individually using unit tests and then system test is applied to test this entire system as one system.

Note

Testing should not be restricted to just execution testing.

Debugging, on the other hand, is an activity that includes execution testing and code correction. The main aim of debugging is locating errors in the program code. Once the errors are located, they are then isolated and fixed to produce an error-free code. Different approaches applied for debugging a code includes:

Brute-Force Method In this technique, a printout of CPU registers and relevant memory locations is taken, studied, and documented. It is the least efficient way of debugging a program and is generally done when all the other methods fail.

Backtracking Method It is a popular technique that is widely used to debug small applications. It works by locating the first symptom of error and then tracing backward across the entire source code until the real cause of error is detected. However, the main drawback of this approach is that with increase in number of source code lines, the possible backward paths become too large to manage.

Cause Elimination In this approach, a list of all possible causes of an error is developed. Then relevant tests are carried out to eliminate each of them. If some tests indicate that a particular cause may be responsible for an error then the data are refined to isolate the error.

Example 1.21

Let us take a problem, collect its requirement, design the solution, implement it in C and then test our program.

Problem Statement To develop an automatic system that accepts marks of a student and generates his/her grade.

Requirements Analysis Ask the users to enlist the rules for assigning grades. These rules are:

Marks	Grade
Above 75	O
60-75	A
50-60	B
40-50	C
Less than 40	D

Design In this phase, write an algorithm that gives a solution to the problem.

Step 1: Enter the marks obtained as M
Step 2: If M > 75 then print "O"
Step 3: If M >= 60 and M < 75 then print "A"
Step 4: If M >= 50 and M < 60 then print "B"
Step 5: If M >= 40 and M < 50 then print "C"
else
print "D"
Step 6: End

Implementation Write the C program to implement the proposed algorithm.

```
#include <stdio.h>
int main()
{  int marks;
```

```
    char grade;
    printf("\n Enter the marks of the student
    : ");
    scanf("%d", &marks);
    if (marks<0 || marks >100)
    {    printf("\n Not Possible");
         exit(1);
    }
    if(marks>=75)
       grade = 'O';
    else if(marks>=60 && marks<75)
       grade = 'A';
    else if(marks>=50 && marks<60)
       grade = 'B';
    else if(marks>=40 && marks<50)
       grade = 'C';
    else
       grade = 'D';
    printf("\n GRADE = %c", grade);
}
```

Test The above program is then tested with different test data to ensure that the program gives correct output for all relevant and possible inputs. The test cases are shown in the table given below.

Test Case ID	Input	Expected Output	Actual Output
1	-12	Not Possible	Not Possible
2	112	Not Possible	Not Possible
3	32	D	D
4	46	C	C
5	54	B	B
6	68	A	A
7	91	O	O
8	40	C	C
9	50	B	B
10	60	A	A
11	75	O	O
12	100	O	O
13	0	D	D

Note in the above table, we have identified test cases for the following,

1."Not Possible" Combinations

2.A middle value from each range

3.Boundary values for each range

POINTS TO REMEMBER

- *Programming* means writing computer programs. While programming, the programmers takes an algorithm and code the instructions in a particular programming language, so that it can be executed by a computer.
- An algorithm provides a blueprint to writing a program to solve a particular problem.
- In the course of processing, data is read from an input device, stored in computer's memory for further processing and then the result of the processing is written to an output device.
- The data is stored in the computer's memory in the form of variables or constants.
- For a complex problem, its algorithm is often divided into smaller units called *functions* (or modules).
- Algorithm validation ensures that the algorithm will work correctly irrespective of the programming language in which it will be implemented.
- An algorithm is analysed to measure its performance in terms of CPU time and memory space required to execute that algorithm.
- Sequence means that each step of the algorithm is executed in the specified order.
- Decision statements are used when the outcome of the process depends on some condition.
- Iteration or Repetition involves executing one or more steps for a number of times, can be implemented using constructs such as the while loops and for loops.
- Programming languages have a vocabulary of syntax and semantics for instructing a computer to perform specific tasks.
- Pseudocode facilitates designers to focus on the logic of the algorithm without getting bogged down by the details of language syntax.
- A structured programming language like C employs a topdown approach in which the overall program structure is broken down into separate modules.
- The entire program or software (collection of programs) development process is divided into a number of phases where each phase performs a well-defined task.

GLOSSARY

Algorithm A step by step instructions that tells the computer how to solve a particular problem.

Backtracking method The technique used to debug small applications which works by locating the first symptom of error and then tracing backward across the entire source code until the real cause of error is detected.

Brute-force method The technique that prints CPU registers and relevant memory locations.

Cause elimination The technique in which list of all possible causes of an error is developed.

Compile errors Errors that occur at the time of compilation of the program.

Debugging An activity that includes execution testing and code correction. The main aim of debugging is locating errors in the program code.

Flowchart A diagrammatic representation that illustrates the sequence of steps that must be performed to solve a problem.

Integration testing A testing technique in which two units that have already been tested are combined into a component and the interface between them is tested.

Linker error Errors that may occur when the linker is not able to find the function definition for a given prototype

Logical errors Errors that result in unexpected and undesirable output which is obviously not correct. Such errors are not detected by the compiler.

Modularization The process of dividing an algorithm into modules/functions.

Program A set of instructions that tells the computer how to solve a particular problem.

Programming The act of writing programs.

Instruction A single operation which when executed convert one state to other.

Programming language A language specifically designed to express computations that can be performed by a computer.

Pseudocode A compact and informal high-level description of an algorithm that uses the structural conventions of a programming language.

Recursion A technique of solving a problem by breaking it down into smaller and smaller sub-problems until you get to a small enough problem that it can be easily solved.

Runtime errors Errors that occur when the program is being executed.

Semantic errors Errors which may comply with rules of the programming language but are not meaningful to the compiler.

Syntax Errors Errors that are generated when rules of C programming language are violated.

System testing A testing technique that checks the entire system.

Testing An activity performed to verify correct behaviour of a program. It is specifically carried out with the intent to find errors.

Unit testing A testing technique applied only on a single unit or module to ensure whether it exhibits the expected behaviour.

EXERCISES

Fill in the blanks

1. A _____ is a set of instructions that tells the computer how to solve a particular problem.
2. Algorithms are mainly used to achieve _____.
3. _____ and _____ statements are used to change the sequence of execution of instructions.
4. _____ is a formally defined procedure for performing some calculation.
5. _____ statements are used when the outcome of the process depends on some condition.
6. Repetition can be implemented using constructs such as _____, _____, and _____.
7. A complex algorithm is often divided into smaller units called _____.
8. The _____ symbol is always the first and the last symbol in a flowchart.
9. _____ is a form of structured English that describes algorithms.
10. _____ is used to express algorithms and as a mode of human communication.
11. The process of dividing an algorithm into modules/functions is called _____.
12. _____ is a technique of solving a problem by breaking it down into smaller and smaller sub-problems until you get to a small enough problem that it can be easily solved.

State True or False

1. An algorithm solves a problem in a finite number of steps.
2. Flowcharts are drawn in the early stages of formulating computer solutions.
3. The main focus of pseudocodes is on the details of the language syntax.
4. Algorithms are implemented using a programming language.
5. Repetition means that each step of the algorithm is executed in a specified order.
6. Terminal symbol depicts the flow of control of the program.
7. Labelled connectors are square in shape.
8. The outputs of each step of an algorithm should be unambiguous. This means that is should be precise.
9. You can have maximum one function in a algorithm.
10. Pseudocode is written using the syntax of a particular programming language.

Multiple Choice Questions

1. Algorithms should be
   (a) precise (b) unambiguous
   (c) clear (d) all of these
2. To check whether a given number is even or odd, you will use which type of control structure?
   (a) sequence (b) decision
   (c) repetition (d) all of these
3. Which one of the following is a graphical or symbolic representation of a process?
   (a) algorithm (b) flowchart
   (c) pseudocode (d) program
4. In a flowchart, which symbol is represented using a rectangle?
   (a) terminal (b) decision
   (c) activity (d) input/output
5. Which of the following details are omitted in pseudocodes?
   (a) variable declaration (b) system specific code
   (c) subroutines (d) all of these
6. A single operation which when executed convert one state to other is called _____.
   (a) instruction (b) program
   (c) software (d) control
7. Algorithm is validated to
   (a) measure its CPU time
   (b) measure the memory consumed
   (c) check if it is correct
   (d) all of these
8. Programming languages have a vocabulary of _____ for instructing a computer to perform specific tasks.
   (a) syntax (b) semantics
   (c) both of these (d) none of these
9. Syntax and semantics of a language are strictly checked in _____.
   (a) algorithm (b) flowchart
   (c) pseudocode (d) program

Review Questions

1. Define an algorithm. How is it useful in the context of software development?
2. Explain and compare the approaches for designing an algorithm.
3. What is modularization?
4. Explain sequence, repetition, and decision statements. Also give the keywords used in each type of statement.
5. With the help of an example, explain the use of a flowchart.
6. How is a flowchart different from an algorithm? Do we need to have both of them for program development?
7. What do you understand by the term pseudocode?
8. Differentiate between algorithm and pseudocodes.
9. Give the characteristics of an algorithm.
10. What do you understand by the term recursion?
11. Write an algorithm and draw a flowchart that calculates salary of an employee. Prompt the user to enter the Basic Salary, HRA, TA, and DA. Add these components to calculate the Gross Salary. Also deduct 10% salary from the Gross Salary to be paid as tax.
12. Draw a flowchart and write an algorithm and a psuedocode for the following problem statements
    (a) Cook maggi
    (b) Cross road
    (c) Calculate bill of items purchased
    (d) To find out whether a number is positive or negative
    (e) Print "Hello" five times on the screen
    (f) Find area of a rectangle
    (g) Convert meters into centimeters
    (h) Find the sum of first 10 numbers