

Session 1: Introduction to Problem Solving and Programming Languages

Learning Objectives

After completing this chapter, you will be able to:

- ☐ Explain the basic concepts of problem solving
- ☐ List the steps involved in program development
- ☐ List the advantages of top-down programming
- ☐ Describe programming languages, their types, and features

Problem Solving Aspect

Problem solving is a creative process. It is an act of defining a problem, determining the cause of the problem, identifying, prioritizing, and selecting alternatives for a solution and implementing a solution.

A problem can be solved successfully only after making an effort to understand the problem. To understand the problem, the following questions help:

- ☐ What do we know about the problem?
- ☐ What is the information that we have to process in order to find the solution?
- ☐ What does the solution look like?
- ☐ What sort of special cases exist?
- ☐ How can we recognize that we have found the solution?

It is important to see if there are any similarities between the current problem and other problems that have already been solved. We have to be sure that the past experience does not hinder us in developing new methodology or technique for solving a problem. The important aspect to be considered in problem-solving is the ability to view a problem from a variety of angles.

There is no universal method for solving a given problem. Different strategies appear to be good for different problems. Some of the well known strategies are:

- ☐ Divide and Conquer
- ☐ Greedy Method
- ☐ Dynamic Programming
- ☐ Backtracking
- ☐ Branch and Bound

Program Development Steps

The various steps involved in Program Development are:

- ☐ Defining or Analyzing the problem
- ☐ Design (Algorithm)
- ☐ Coding
- ☐ Documenting the program
- ☐ Compiling and Running the Program
- ☐ Testing and Debugging
- ☐ Maintenance

Analyzing or Defining the Problem

The problem is defined by doing a preliminary investigation. Defining a problem helps us to understand the problem clear. It is also known as Program Analysis.

Tasks in defining a problem:

- ☐ Specifying the input requirements
- ☐ Specifying the output requirements
- ☐ Specifying the processing requirements

Specifying the input requirements

Determine the inputs required and source of the data. The input specification is obtained by answering the following questions:

- ☐ What specific values will be provided as input to the program?
- ☐ What format will the values be?
- ☐ For each input item, what is the valid range of values that it may assume?
- ☐ What restrictions are placed on the use of these values?

Specifying the output requirements

Describe in detail the output that will be produced. The output specification is obtained by answering the following questions:

- ☐ What values will be produced?
- ☐ What is the format of these values?
- ☐ What specific annotation, headings, or titles are required in the report?
- ☐ What is the amount of output that will be produced?

Specifying the Processing Requirements

Determine the processing requirements for converting the input data to output. The processing requirement specification is obtained by answering the following questions:

- ☐ What is the method (technique) required in producing the desired output?
- ☐ What calculations are needed?
- ☐ What are the validation checks that need to be applied to the input data?

Example 1.1 Find the factorial of a given number

Input: Positive valued integer number

Output: Factorial of that number

Process: Solution technique which transforms input into output. Factorial of a number can be calculated by the formula $n! = 1*2*3*4*...*n$

Design

A design is the path from the problem to a solution in code. Program Design is both a product and a process. The process results in a theoretical framework for describing the effects and consequences of a program as they are related to its development and implementation.

A well designed program is more likely to be:

- ☐ Easier to read and understand later
- ☐ Less of bugs and errors
- ☐ Easier to extend to add new features
- ☐ Easier to program in the first place

Modular Design

Once the problem is defined clearly, several design methodologies can be applied. An important approach is Top-Down programming design. It is a structured design technique which breaks up the problem into a set of sub-problems called Modules and creates a hierarchical structure of modules.

While applying top-down design to a given problem, consider the following guidelines:

- ☐ A problem is divided it into smaller logical sub-problems, called Modules
- ☐ Each module should be independent and should have a single task to do
- ☐ Each module can have only one entry point and one exit point, so that the logic flow of the program is easy to follow
- ☐ When the program is executed, it must be able to move from one module to the next in sequence, until the last module is executed
- ☐ Each module should be of manageable size, in order to make the design and testing easier

Top-down design has the following advantages:

- ☐ Breaking up the problem into parts helps us to clarify what is to be done
- ☐ At each step of refinement, the new parts become more focussed and, therefore, easier to design
- ☐ Modules may be reused
- ☐ Breaking the problem into parts allows more than one person to work on the solution simultaneously

Algorithm (Developing a Solution technique)

An algorithm is a step-by-step description of the solution to a problem. It is defined as an ordered sequence of well-defined and effective operations which, when carried out for a given set of initial conditions, produce output, and terminate in a finite time. The term “ordered sequence” specifies, after the completion of each step in the algorithm, the next step must be unambiguously defined.

An algorithm must be:

- ☐ Definite
- ☐ Finite
- ☐ Precise and Effective
- ☐ Implementation independent (only for problem not for programming languages)

Developing Algorithms

Algorithm development process is a trial-and-error process. Programmers make initial attempt to the solution and review it, to test its correctness. The errors identified leads to insertions, deletions, or modifications to the existing algorithm.

This refining continues until the programmer is satisfied that, the algorithm is essentially correct and ready to be executed. The more experience we gain in developing an algorithm, the closer our first attempt will be to a correct solution and the less revision will be required. However, a novice programmer should not view developing algorithm as a single-step operation.

Example 1.2: Algorithm for finding factorial of a given number

Step 1: Start
Step 2: Initialize factorial to be 1, i to be 1
Step 3: Input a number n
Step 4: Check whether the number is 0. If so report factorial is 1 and goto step 9
Step 5: Repeat step 6 through step 7 n times
Step 6: Calculate factorial = factorial * i
Step 7: Increment i by 1
Step 8: Report the calculated factorial value
Step 9: Stop

Pseudo Code

Pseudo code is an informal high-level description of an algorithm that uses the structural conventions of programming languages, but omits language-specific syntax. It is an outline of a program written in English or the user's natural language.

Example 1.3: Pseudo Code for finding factorial of a given number

Step 1: START
Step 2: DECLARE the variables n, fact, i
Step 2: SET variable fact =1 and i =1
Step 3: READ the number n


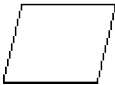







Step 4: IF n = 0 then



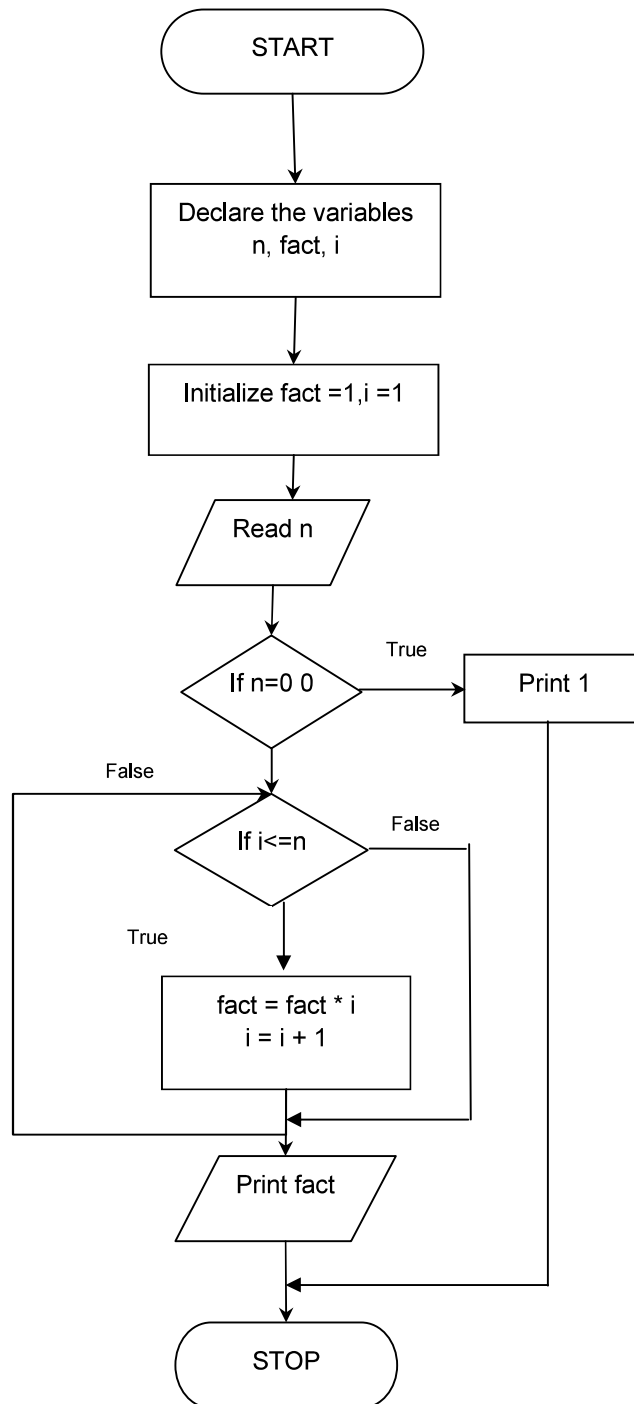
Step 4.1: PRINT factorial = 1
Step 4.2: GOTO Step 9
Step 5: WHILE the condition $i \leq n$ is true, repeat Step 6 through Step 7
Step 6: COMPUTE $fact = fact * i$
Step 7: INCREMENT i by 1
Step 8: PRINT the factorial value
Step 9: STOP

Flowchart

Flowchart is a diagrammatic representation of an algorithm. It uses different symbols to represent the sequence of operations, required to solve a problem. It serves as a blueprint or a logical diagram of the solution to a problem. Typical flowchart symbols are given below:

	Represents Start, End
	Represents Input, Output data
	Represents Process (actions, calculations)
	Represents Decision Making
	Represents Pre-defined Process / module
	Represents off page connector which are used to indicate that the flow chart continues on another page. Page numbers are usually placed inside for easy reference.
	Connector Symbol represents the exit to, or entry from, another part of the same flow chart. It is usually used to break a flow line that will be continued elsewhere.
	The Document Symbol is used to represent any type of hard copy input or output (i.e. reports).
	Represents control flow

Example 1.4: Flow Chart for finding factorial of a given number



Coding

An algorithm expressed in programming languages is called Program. Writing a program is called Coding. The logic that has been developed in the algorithm is used to write the program.

Documenting the Program

Documentation explains how the program works and how to use the program. Documentation can be of great value, not only to those involved in maintaining or modifying a program, but also to the programmers themselves. Details of particular programs, or particular pieces of programs, are easily forgotten or confused without suitable documentation.

Documentation comes in two forms:

- ❑ External documentation, which includes things such as reference manuals, algorithm descriptions, flowcharts, and project workbooks
- ❑ Internal documentation, which is part of the source code itself (essentially, the declarations, statements, and comments)

Compiling and Executing the Program

Compilation is a process of translating a source program into machine understandable form. The compiler is system software, which does the translation after examining each instruction for its correctness. The translation results in the creation of object code.

After compilation, Linking is done if necessary. Linking is the process of putting together all the external references (other program files and functions) that are required by the program. The program is now ready for execution.

During execution, the executable object code is loaded into the computer's memory and the program instructions are executed.

Testing

Testing is the process of executing a program with the deliberate intent of finding errors. Testing is needed to check whether the expected output matches the actual output. Program should be tested with all possible input data and control conditions.

Testing is done during every phase of program development. Initially, requirements can be tested for its correctness. Then, the design (algorithm, flow charts) can be tested for its exactness and efficiency. Structured walk through is made to verify the design.

Programs are tested with several test criteria and the important ones are given below:

- ❑ Test whether each and every statement in the program is executed at least once (Basic path testing)
- ❑ Test whether every branch in the program is traversed at least once (control flow)
- ❑ Test whether the input data flows through the program and is converted to an output (data flow)

The probability of discovering errors through testing can be increased by selecting significant test cases. It is important to design test cases for abnormal input conditions.

The Boundary (or Extreme) Cases

How does the algorithm perform at the extremes of the valid cases?

The Unusual Cases

What happens when the input data violates the normal conditions of the problem or represent unusual condition?

The Invalid Cases

How does the algorithm react for data which are patently illegal or completely meaningless?

An algorithm should work correctly and produce meaningful results for any data. This is called foolproof programming.

Debugging

Debugging is a process of correcting the errors. Programs may have logical errors which cannot be caught during compilation. Debugging is the process of identifying their root causes. One of the ways to ensure the correctness of the program is by printing out the intermediate results at strategic points of computation.

Some programmers use the terms “testing” and “debugging” interchangeably, but careful programmers distinguish between the two activities. Testing means detecting errors. Debugging means diagnosing and correcting the root causes.

On some projects, debugging occupies as much as 50 percent of the total development time. For many programmers, debugging is the hardest part of programming because of improper documentation.

Maintenance

Programs require a continuing process of maintenance and modification to keep pace with changing requirements and implementation technologies. Maintainability and modifiability are essential characteristics of every program. Maintainability of the program is achieved by:

- ❑ Modularizing it
- ❑ Providing proper documentation for it
- ❑ Following standards and conventions (naming conventions, using symbolic constants etc)

Introduction to Programming Languages

What is a Programming Language?

Computer Programming is an art of making a computer to do the required operations, by means of issuing sequence of commands to it.

A programming language can be defined as a vocabulary and set of grammatical rules for instructing the computer to perform specific tasks. Each programming language has a unique set of characters, keywords and the syntax for organizing programming instructions.

The term programming languages usually refers to high-level languages, such as BASIC, C, C++, COBOL, FORTRAN, Ada, and Pascal.



Why Study Programming Languages?

The design of new programming languages and implementation methods have been evolved and improved to meet the change in requirements. Thus, there are many new languages.

The study of more than one programming language helps us:

- ❑ to master different programming paradigms
- ❑ to enhance the skills to state different programming concepts
- ❑ to understand the significance of a particular language implementation
- ❑ to compare different languages and to choose appropriate language
- ❑ to improve the ability to learn new languages and to design new languages

Types and Categories of Programming Languages

Types of Programming Languages

There are two major types of programming languages:

- ❑ Low Level Languages
- ❑ High Level Languages

Low Level Languages

The term *low level* refers closeness to the way in which the machine has been built. Low level languages are machine oriented and require extensive knowledge of computer hardware architecture and its configuration. Low Level languages are further divided in to Machine language and Assembly language.

(a) Machine Language

Machine Language is the only language that is directly understood by the computer. It does not need any translator program. The instructions are called machine instruction (machine code) and it is written as strings of 1's (one) and 0's (zero). When this sequence of codes is fed in to the computer, it recognizes the code and converts it in to electrical signals.

For example, a program instruction may look like this: 1011000111101

Machine language is considered to be the first generation language. Because of it design, machine language is not an easy language to learn. It is also difficult to debug the program written in this language.

Advantage

- ❑ The program runs faster because no translation is needed. (It is already in machine understandable form)

Disadvantages

- ❑ It is very difficult to write programs in machine language. The programmer has to know details of hardware to write program
- ❑ It is difficult to debug the program

(b) Assembly Language

In assembly language, set of mnemonics (symbolic keywords) are used to represent machine codes. Mnemonics are usually combination of words like ADD, SUB and LOAD etc. In order to execute the programs written in assembly language, a translator program is required to translate it

to the machine language. This translator program is called *Assembler*. Assembly language is considered to be the second-generation language.

Advantages:

- ❑ The symbolic keywords are easier to code and saves time and effort
- ❑ It is easier to correct errors and modify programming instructions
- ❑ Assembly Language has utmost the same efficiency of execution as the machine level language, because there is one-to-one translation between assembly language program and its corresponding machine language program

Disadvantages:

- ❑ Assembly languages are machine dependent. A program written for one computer might not run in other computer.

High Level Languages

High level languages are the simple languages that use English like instructions and mathematical symbols like +, -, %, /, for its program construction. In high level languages, it is enough to know the logic and required instructions for a given problem, irrespective of the type of computer used.

Compiler is a translator program which converts a program in high level language in to machine language.

Higher level languages are problem-oriented languages because the instructions are suitable for solving a particular problem.

For example, COBOL (Common Business Oriented Language) is mostly suitable for business oriented applications. There are some numerical & mathematical oriented languages like FORTRAN (Formula Translation) and BASIC (Beginners All-purpose Symbolic Instruction Code).

Advantages of High Level Languages

- ❑ High level languages are easy to learn and use

Categories of programming languages

Numerical Languages

Early computer technology dates from the era just before World War 2 in the late 1930s to the early 1940s. These early machines were designed to solve numerical problems and were thought of as ELECTRONIC CALCULATORS. Numerical calculations were the dominant form of application for these early machines.

Business Languages

Business data processing was an early application domain developed after numerical applications. In 1959, the US department of Defense sponsored a meeting to develop COMMON BUSINESS LANGUAGE (CBL), which would be a business-oriented language that used English as much as possible for its notation. This, in turn, led to the formation of a Short Range Committee to develop COBOL.

Artificial Intelligence Languages (AI)

The first step towards the development of AI languages commenced with the evolution of IPL (Information Processing Language) by the Rand Corporation. The major breakthrough occurred, when John McCarthy of MIT designed LISP (List Processing) for the IBM 704. Later, more AI languages like SNOBOL & PROLOG were designed.

Systems Languages

Because of the need of efficiency, the use of assembly language held on for years in the system area long after other application domains started to use higher-level languages. Many systems programming languages such as CPL & BCPL were designed, though not widely used. The major landmark here is the development of UNIX, where high level languages also proceed to work effectively.

What makes a Good Language?

Every language has its strengths and weaknesses.

For example, FORTRAN is a particularly good language for processing numerical data, but it does not lend itself very well to organize large programs. PASCAL is very good for writing well-structured and readable programs, but it is not as flexible as the C programming language. C++ embodies powerful object-oriented features, but it is complex and difficult to learn. The choice of which language to use depends on the type of computer used, type of program, and the expertise of the programmer.

Following are the most important features that would make a programming language efficient and easy to use:

Clarity, Simplicity and Unity: A programming Language provides, both a conceptual framework for thinking about algorithms and a means for expressing these algorithms. The syntax of a language should be such that programs may be written, tested and maintained with ease.

Orthogonality: This refers to the attribute of being able to combine various features of a language in all possible combinations, with every combination being meaningful. Orthogonality makes a language easy to learn and write programs, because there are fewer exceptions & special cases to remember.

Naturalness for the application: A language needs syntax that when properly used allows the program structure to reflect the underlying logical structure of the algorithm. The language should provide appropriate data structures, operations, control structures and natural syntax for the problem to be solved.

Support for abstraction: Even with the most natural programming language for an application, there is always a substantial gap remaining between the abstract data structures & operations that characterize the solution to a problem and the particular data structures and operations built into a language.

Portability of Programs: Portability is an important criterion for many programming projects which essentially indicates the transportability of the resulting programs from the computer on which they are developed to other computer systems. A language whose definition is independent of the features of a particular machine forms a useful base for the production of transportable programs.

Cost of use: Cost of use is measured on different languages like:

- ❑ **Cost of program execution:** Optimizing compilers, efficient register allocation, design of efficient run-time support mechanisms are all factors that contribute towards cost of program execution. This is highly critical for large programs that will be executed continuously.
- ❑ **Cost of Program creation, testing & use:** This implies design, coding, testing, usage & maintenance solutions for a problem with minimum investment of programmer time & energy.

Cost of Program Maintenance: The highest cost involved in any program is the total life-cycle costs including development costs & the cost of maintenance of the program while it is in production use.

Program Development Environments

The environment under which a program is designed, coded, tested & debugged is called *Host Environment*. The external environment which supports the execution of a program is termed as *Operating or Target Environment*. Host and Target environment may be different for a program or application.

Programming Environments (Host Environment)

It is the environment in which programs are created and tested. It tends to have less influence on language design than the operating environment in which programs are expected to be executed. The production of programs that operate reliably and efficiently is made much simpler by a good programming environment and by a language that allows the use of good programming tools and practices.

Target Environments

Target environments can be classified into 3 categories – Batch Processing Environment, Interactive Environment, and Embedded System Environment. Each poses different requirement on languages adapted for those environments.

Batch-Processing Environments

In batch-processing environments, the input data are collected in 'batches' on files and are processed in batches by the program. For example, the backup process on an organization. The transaction details of all the departments are collected for backup at one place and the backup is done at a time at the end of the day.

Interactive Environments

In interactive environment, a program interacts directly with a user at a display console, by alternately sending output to the display & receiving input from the keyboard or mouse. Examples include database management systems, word processing systems etc.

Embedded System Environments

An embedded computer system is used to control part of a larger system such as an industrial plant (computerized machineries) or an aircraft. The computer system will be an integral part of the larger system, failure of which would imply failure of the larger system as well.

Summary

- ❑ Program development life cycle involves analysis, algorithm development, coding, documenting, compiling and running, testing, debugging, and maintenance.
- ❑ Top-down program design, divides the problem into smaller logical sub problems, called Modules.
- ❑ An algorithm is a sequence of unambiguous instructions for solving a problem.
- ❑ A programming language is a vocabulary and set of grammatical rules for instructing a computer to perform specific tasks.
- ❑ Two major types of programming languages are Low Level Languages and High Level Languages.
- ❑ The environment under which a program is designed, coded, tested & debugged is called Host environment (programming environment)
- ❑ The environment under which a program is executed is called Target environment.
- ❑ Target environments can be classified into 3 categories.
 - Batch processing environment
 - Interactive environment
 - Embedded System environment

Test your Understanding

1. Represent the following problem in top-down design.
 - ❑ Planning a tour.
2. Give the algorithm, pseudo code and flowchart for the following problem:
 - ❑ Sort a list of numbers in ascending order.
3. Distinguish between testing and debugging.
4. State whether the following is True or False :
 - a. Assembly language is a second generation language.
 - b. Programs written in high Level languages needs translation for executing them.

5. What is meant by portability of programs?
- a. Easy to carry from place to place
 - b. Transportability of resulting program within machine folders
 - c. It can run on any machine
 - d. The program needs to be compiled in every machine

Answers:

3. Testing is to find errors in programs and debugging is to correct their root causes
4. True, True
5. c (it can run on any machine)