

Programming in C

Second Edition

OXFORD
UNIVERSITY PRESS

C

Brief Contents

Preface to the Second Edition	iii
Preface to the First Edition	v
1. Introduction to Programming, Algorithms and Flowcharts	1
2. Basics of C	39
3. Input and Output	94
4. Control Statements	117
5. Arrays and Strings	169
6. Functions	214
7. Pointers in C	268
8. User-defined Data Types and Variables	350
9. Files in C	388
10. Linked Lists	423
11. Advanced C	460
12. Stacks, Queues, and Trees	492
Appendices	524
Bibliography and References	544
Index	545

C

Contents

<i>Preface to the Second Edition</i>	iii
<i>Preface to the First Edition</i>	v

1 INTRODUCTION TO PROGRAMMING, ALGORITHMS AND FLOWCHARTS **1**

1.1 Programs and Programming	1
<i>System Software</i>	2
<i>Application Software</i>	2
1.2 Programming Languages	2
<i>System Programming Languages</i>	3
<i>Application Programming Languages</i>	3
<i>Low-level Languages</i>	3
<i>High-level Languages</i>	5
1.3 Compiler, Interpreter, Loader, and Linker	6
<i>Compiling and Executing High-level Language Programs</i>	6
<i>Linker</i>	7
<i>Loader</i>	7
<i>Linking Loader and Linkage Editor</i>	8
1.4 Program Execution	8
1.5 Fourth Generation Languages	9
1.6 Fifth Generation Languages	10
1.7 Classification of Programming	10
<i>Procedural Languages</i>	10
<i>Problem-oriented Languages</i>	11
<i>Non-procedural Languages</i>	11
1.8 Structured Programming Concept	11
<i>Top-down Analysis</i>	12

<i>Modular Programming</i>	12
<i>Structured Code</i>	13
<i>The Process of Programming</i>	13
1.9 Algorithms	14
<i>What is an Algorithm?</i>	14
<i>Different Ways of Stating Algorithms</i>	14
<i>Key Features of an Algorithm and the Step-form</i>	14
<i>What are Variables?</i>	16
<i>Subroutines</i>	17
<i>Strategy for Designing Algorithms</i>	30
<i>Tracing an Algorithm to Depict Logic</i>	31
<i>Specification for Converting Algorithms into Programs</i>	32

2 BASICS OF C **39**

2.1 Introduction	39
<i>Why Learn C?</i>	40
<i>The Future of C</i>	40
2.2 Standardizations of C Language	40
2.3 Developing Programs In C	41
2.4 A Simple C Program	45
2.5 Parts of C Program Revisited	47
2.6 Structure of a C Program	48
2.7 Concept of a Variable	49

2.8 Data Types in C 50
 2.9 Program Statement 55
 2.10 Declaration 56
 2.11 How Does The Computer Store Data in Memory? 57
 How Integers are Stored? 57
 How Floats and Doubles are Stored? 58
 2.12 Token 60
 Identifier 60
 Keywords 61
 Constant 61
 Assignment 63
 Initialization 64
 2.13 Operators and Expressions 65
 Arithmetic Operators in C 66
 Relational Operators in C 71
 Logical Operators in C 71
 Bitwise Operators in C 72
 Conditional Operator in C 73
 Comma Operator 73
 Sizeof Operator 74
 Expression Evaluation—Precedence and Associativity 74
 2.14 Expressions Revisited 77
 2.15 Lvalues and Rvalues 77
 2.16 Type Conversion in C 78
 Type Conversion in Expressions 78
 Conversion by Assignment 79
 Casting Arithmetic Expressions 81
 2.17 Working with Complex Numbers 86

3 INPUT AND OUTPUT 94

3.1 Introduction 94
 3.2 Basic Screen and Keyboard I/O in C 95
 3.3 Non-Formatted Input and Output 96
 Single Character Input and Output 96
 Single Character Input 96
 Single Character Output 96
 Additional Single Character Input and Output Functions 97
 3.4 Formatted Input and Output Functions 100
 Output Function printf() 100
 Input Function scanf() 106

4 CONTROL STATEMENTS 117

4.1 Introduction 117
 4.2 Specifying Test Condition Forselection and Iteration 119

4.3 Writing Test Expression 119
 Understanding How True and False is Represented in C 120
 4.4 Conditional Execution and Selection 124
 Selection Statements 124
 The Conditional Operator 131
 The Switch Statement 133
 4.5 Iteration and Repetitive Execution 137
 While Construct 138
 For Construct 143
 do-while Construct 151
 4.6 Which Loop Should be Used? 153
 Using Sentinel Values 153
 Using Prime Read 154
 Using Counter 155
 4.7 Goto Statement 155
 4.8 Special Control Statements 156
 4.9 Nested Loops 159

5 ARRAYS AND STRINGS 169

5.1 Introduction 169
 5.2 One-Dimensional Array 170
 Declaration of a One-dimensional Array 171
 Initializing Integer Arrays 173
 Accessing Array Elements 173
 Other Allowed Operations 174
 Internal Representation of Arrays in C 176
 Variable Length Arrays and the C99 changes 177
 Working with One-dimensional Array 177
 5.3 Strings: One-dimensional Character Arrays 182
 Declaration of a String 182
 String Initialization 182
 Printing Strings 183
 String Input 184
 Character Manipulation in the String 190
 String Manipulation 191
 5.4 Multidimensional Arrays 199
 Declaration of a Two-dimensional Array 199
 Declaration of a Three-dimensional Array 199
 Initialization of a Multidimensional Array 199
 Unsize Array Initializations 201
 Accessing Multidimensional Arrays 201
 Working with Two-dimensional Arrays 202
 5.5 Arrays of Strings: Two-dimensional Character Array 206
 Initialization 206
 Manipulating String Arrays 206

6 FUNCTIONS 214

- 6.1 Introduction 214
- 6.2 Concept of Function 215
 - Why are Functions Needed?* 215
- 6.3 Using Functions 216
 - Function Prototype Declaration* 216
 - Function Definition* 217
 - Function Calling* 219
- 6.4 Call by Value Mechanism 221
- 6.5 Working with Functions 221
- 6.6 Passing Arrays to Functions 224
- 6.7 Scope and Extent 227
 - Concept of Global and Local Variables* 227
 - Scope Rules* 229
- 6.8 Storage Classes 231
 - Storage Class Specifiers for Variables* 231
 - Storage Class Specifiers for Functions* 234
 - Linkage* 234
- 6.9 The Inline Function 234
- 6.10 Recursion 235
 - What is Needed for Implementing Recursion?* 235
 - How is Recursion Implemented?* 239
 - Comparing Recursion and Iteration* 241
- 6.11 Searching and Sorting 241
 - Searching Algorithms* 241
 - Sorting Algorithms* 243
- 6.12 Analysis of Algorithms 248
 - Asymptotic Notation* 250
 - Efficiency of Linear Search* 252
 - Binary Search Analysis* 253
 - Analysis of Bubble Sort* 254
 - Analysis of Quick Sort* 255
 - Disadvantages of Complexity Analysis* 255

7 POINTERS IN C 268

- 7.1 Introduction 268
- 7.2 Understanding Memory Addresses 269
- 7.3 Address Operator (&) 271
- 7.4 Pointer 272
 - Declaring a Pointer* 272
 - Initializing Pointers* 274
 - Indirection Operator and Dereferencing* 276
- 7.5 Void Pointer 278
- 7.6 Null Pointer 278
- 7.7 Use of Pointers 279
- 7.8 Arrays and Pointers 282
 - One-dimensional Arrays and Pointers* 282
 - Passing an Array to a Function* 285
 - Differences Between Array Name and Pointer* 286

- 7.9 Pointers and Strings 288
- 7.10 Pointer Arithmetic 289
 - Assignment* 290
 - Addition or Subtraction with Integers* 291
 - Subtraction of Pointers* 298
 - Comparing Pointers* 299
- 7.11 Pointers to Pointers 300
- 7.12 Array of Pointers 302
- 7.13 Pointers To an Array 306
- 7.14 Two-dimensional Arrays and Pointers 307
 - Passing Two-dimensional Array to a Function* 309
- 7.15 Three-dimensional Arrays 316
- 7.16 Pointers to Functions 317
 - Declaration of a Pointer to a Function* 317
 - Initialization of Function Pointers* 317
 - Calling a Function using a Function Pointer* 317
 - Passing a Function to another Function* 318
 - How to Return a Function Pointer* 319
 - Arrays of Function Pointers* 320
- 7.17 Dynamic Memory Allocation 320
 - Dynamic Allocation of Arrays* 323
 - Freeing Memory* 325
 - Reallocating Memory Blocks* 327
 - Implementing Multidimensional Arrays using Pointers* 328
- 7.18 Offsetting a Pointer 331
- 7.19 Memory Leak and Memory Corruption 333
- 7.20 Pointer and Const Qualifier 334
 - Pointer to Constant* 334
 - Constant Pointers* 335
 - Constant Parameters* 335

8 USER-DEFINED DATA TYPES AND VARIABLES 350

- 8.1 Introduction 350
- 8.2 Structures 351
 - Declaring Structures and Structure Variables* 351
 - Accessing the Members of a Structure* 354
 - Initialization of Structures* 355
 - Copying and Comparing Structures* 359
 - Typedef and its Use in Structure Declarations* 361
 - Nesting of Structures* 362
 - Arrays of Structures* 363
 - Initializing Arrays of Structures* 364
 - Arrays within the Structure* 365
 - Structures and Pointers* 365
 - Structures and Functions* 367
- 8.3 Union 370
 - Declaring a Union and its Members* 370

<i>Accessing and Initializing the Members of a Union</i>	371	10.5 Introduction to Circular Doubly Linked List	450
<i>Structure Versus Union</i>	372	10.6 Applications of Linked Lists	451
8.4 Enumeration Types	373	<i>Dynamic Storage Management</i>	451
8.5 Bitfields	374	<i>Garbage Collection and Compaction</i>	452
9 FILES IN C	388	10.7 Disadvantages of Linked Lists	454
9.1 Introduction	388	10.8 Array Versus Linked List Revisited	454
9.2 Using Files in C	390	11 ADVANCED C	460
<i>Declaration of File Pointer</i>	390	11.1 Introduction	460
<i>Opening a File</i>	391	11.2 Bitwise Operator	461
<i>Closing and Flushing Files</i>	392	<i>Bitwise and</i>	462
9.3 Working with Text Files	393	<i>Bitwise or</i>	463
<i>Character Input and Output</i>	393	<i>Bitwise Exclusive-OR</i>	464
<i>End of File (EOF)</i>	394	<i>Bitwise Not</i>	464
<i>Detecting the End of a File Using the feof() Function</i>	400	<i>Bitwise Shift Operator</i>	465
9.4 Working with Binary Files	401	11.3 Command-Line Arguments	467
9.5 Direct File Input and Output	402	11.4 The C Preprocessor	470
<i>Sequential Versus Random File Access</i>	403	<i>The C Preprocessor Directives</i>	470
9.6 Files of Records	403	<i>Predefined Identifiers</i>	474
<i>Working with Files of Records</i>	403	11.5 Type Qualifier	475
9.7 Random Access to Files of Records	410	<i>Const Qualifier</i>	476
9.8 Other File Management Functions	413	<i>Volatile Qualifier</i>	478
<i>Deleting a File</i>	413	<i>Restrict Qualifier</i>	479
<i>Renaming a File</i>	413	11.6 Variable Length Argument List	480
9.9 Low-Level I/O	414	11.7 Memory Models and Pointers	481
10 LINKED LISTS	423	12 STACKS, QUEUES, AND TREES	492
10.1 Introduction	423	12.1 Introduction	492
10.2 Singly Linked List	425	12.2 Stack	493
<i>Insertion of a Node in a Singly Linked List</i>	430	<i>Implementation of Stack</i>	493
<i>Deletion of a Node from a Singly Linked List</i>	434	<i>Application of Stack</i>	498
<i>Sorting a Singly Linked List</i>	435	12.3 Queue	499
<i>Destroying a Singly Linked List</i>	436	<i>Implementation of Queue</i>	499
<i>More Complex Operations on Singly Linked Lists</i>	437	<i>Other Variations of Queue</i>	505
10.3 Circular Linked Lists	440	<i>Applications of Queue</i>	505
<i>Appending a Node</i>	441	12.4 Tree	506
<i>Displaying a Circular Linked List</i>	442	<i>Some Basic Tree Terminology</i>	507
<i>Inserting a Node After a Specified Node</i>	442	<i>Binary Tree</i>	507
<i>Inserting a Node Before a Particular Node</i>	443	<i>Traversals of a Binary Tree</i>	509
<i>Deleting a Node</i>	444	<i>Kinds of Binary Trees</i>	511
<i>Sorting a Circular Linked List</i>	446	<i>Binary Search Tree</i>	511
10.4 Doubly Linked List	446	<i>Application of Tree</i>	518
<i>Operations on Doubly Linked List</i>	447	Appendices	524
<i>Advantages/Disadvantages of Doubly Linked Lists</i>	450	Bibliography and References	544
		Index	545

C

Introduction to Programming, Algorithms and Flowcharts

Chapter 1

Learning Objectives

After reading this chapter, the readers will be able to

- define program and programming
- identify system programs and application programs
- get a basic concept of high-, middle-, and low-level languages
- briefly understand compiler, interpreter, linker, and loader functions
- understand algorithms and the key features of an algorithm—sequence, decision, and repetition
- learn the different ways of stating algorithms—step-form, flowchart, etc.
- define variables, types of variables, and naming conventions for variables
- decide a strategy for designing algorithms

1.1 PROGRAMS AND PROGRAMMING

A computer can neither think nor make a decision on its own. In fact, it is not possible for any computer to independently analyze a given data and find a solution on its own. It needs a program which will convey what is to be done. A program is a set of logically related instructions that is arranged in a sequence that directs the computer in solving a problem.

The process of writing a program is called programming. It is a necessary and critical step in data processing. An incorrect program delivers results that cannot be used. There are two ways by which one can acquire a program—either purchase an existing program, referred to as *packaged software* or prepare a new program from scratch, in which case it is called *customized software*.

Computer software can be broadly classified into two categories: system software and application software.

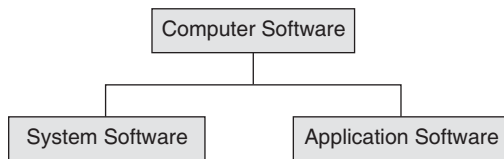


Figure 1.1 Computer software classification

1.1.1 System Software

System software is a collection of programs that interfaces with the hardware. Some common categories of system software are described as follows.

Language translator It is a system software that transforms a computer program written by a user into a form that can be understood by the machine.

Operating system (OS) This is the most important system software that is required to operate a computer system. An operating system manages the computer's resources effectively, takes care of scheduling multiple jobs for execution, and manages the flow of data and instructions between the input/output units and the main memory. An operating system has become a part of computer software with the advent of the third generation computers. Since then a number of operating systems have been developed and some have undergone several revisions and modifications to achieve better utilization of computer resources. Advances in computer hardware have helped in the development of more efficient operating systems.

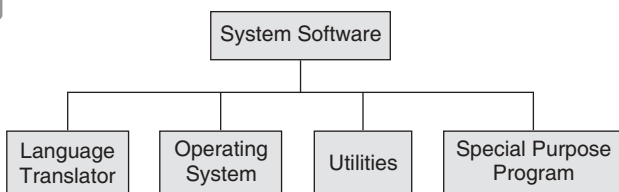


Figure 1.2 Categories of system software

1.1.2 Application Software

Application software is written to enable the computer to solve a specific data processing task. There are two categories of application software: pre-written software packages and user application programs.

A number of powerful application software packages that do not require significant programming knowledge have been developed. These are easy to learn and use compared to programming languages. Although these packages can perform many general and special functions, there are applications where these packages are found to be inadequate. In such cases, user application programs are written to meet the exact requirements. A user application program may be written using one of these packages or a programming language. The most important categories of software packages available are

- Database management software
- Spreadsheet software
- Word processing, Desktop Publishing (DTP), and presentation software
- Multimedia software
- Data communication software
- Statistical and operational research software

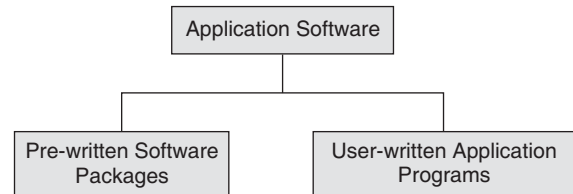


Figure 1.3 Categories of application software

Points to Note

1. A program is a sequence of logically related instructions and the process of making it is programming.
2. A program is a software that is broadly categorized as system software and application software.

1.2 PROGRAMMING LANGUAGES

To write a computer program, a standard programming language is used. A programming language is composed of a set of instructions in a language understandable to the programmer and recognizable by a computer. Programming languages can be classified as high-level, middle-level, and low-level. High-level languages such as BASIC, COBOL (Common Business Oriented Programming Language), and FORTRAN (Formula Translation Language) are used

to write application programs. A middle-level language such as C is used for writing application and system programs. A low-level language such as the assembly language is mostly used to write system programs.

Low-level programming languages were the first category of programming languages to evolve. Gradually, high-level and middle-level programming languages were developed and put to use.

Figure 1.4 depicts the growth in computer languages since the 1940s. The figure is meant to give some idea of the times that the different generations appeared, time scales, and relativity of computer languages to each other and the world of problem solving.

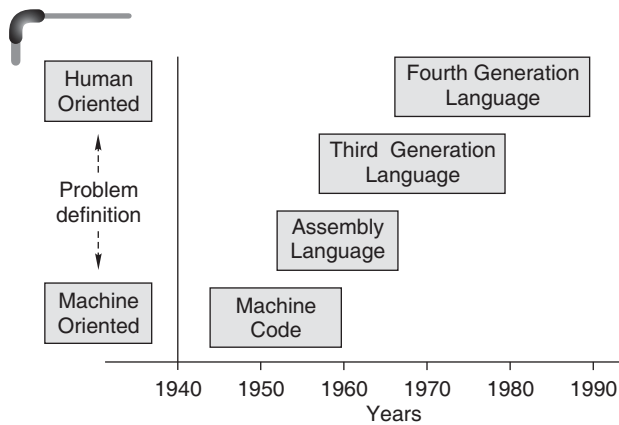


Figure 1.4 Growth of computer languages

1.2.1 System Programming Languages

System programs or softwares are designed to make the computer easier to use. An example of system software is an operating system consisting of many other programs that control input/output devices, memory, processor, schedule the execution of multiple tasks, etc. To write an operating system program, the programmer needs instructions to control the computer's circuitry as well as manage the resources of the computer. For example, instructions that move data from one location of storage to a register of the processor are required. Assembly language, which has a one-to-one correspondence with machine code, was the normal choice for writing system programs like operating systems. But, today C is widely used to develop system software.

1.2.2 Application Programming Languages

There are two main categories of application programs: *business programs* and *scientific application programs*. Application programs are designed for specific computer applications, such as payroll processing and inventory control. To write programs for payroll processing or other such applications, the programmer does not need to control the basic circuitry of a computer. Instead, the programmer needs instructions that make it easy to input data, produce output, perform calculations, and store and retrieve data. Programming languages suitable for such application programs have the appropriate instructions. Most programming languages are designed to be good for one category of applications but not necessarily for the other, although there are some general-purpose languages that support both types. Business applications are characterized by processing of large inputs and high-volume data storage and retrieval but call for simple calculations. Languages which are suitable for business program development must support high-volume input, output, and storage but do not need to support complex calculations. On the other hand, programming languages designed for writing scientific programs contain very powerful instructions for calculations but have poor instructions for input, output, etc. Among the traditionally used programming languages, COBOL is more suitable for business applications whereas FORTRAN is more suitable for scientific applications.

1.2.3 Low-level Languages

A low-level computer programming language is one that is closer to the native language of the computer, which is 1's and 0's.

Machine language

This is a sequence of instructions written in the form of binary numbers consisting of 1's and 0's to which the computer responds directly. The machine language is also referred to as the machine code, although the term is used more broadly to refer to any program text.

A machine language instruction generally has three parts as shown in Fig. 1.5. The first part is the command or operation code that conveys to the computer what function

has to be performed by the instruction. All computers have operation codes for functions such as adding, subtracting and moving. The second part of the instruction either specifies that the operand contains data on which the operation has to be performed or it specifies that the operand contains a location, the contents of which have to be subjected to the operation.

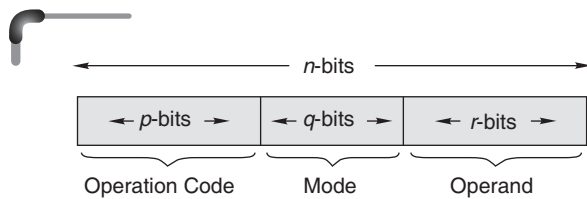


Figure 1.5 General format of machine language instruction

Just as hardware is classified into generations based on technology, computer languages also have a generation classification based on the level of interaction with the machine. Machine language is considered to be the *first generation language* (1GL).

Advantage of machine language The CPU directly understands machine instructions, and hence no translation is required. Therefore, the computer directly starts executing the machine language instructions, and it takes less execution time.

Disadvantages of machine language

- **Difficult to use** It is difficult to understand and develop a program using machine language. For anybody checking such a program, it would be difficult to forecast the output when it is executed. Nevertheless, computer hardware recognizes only this type of instruction code.
- **Machine dependent** The programmer has to remember machine characteristics while preparing a program. As the internal design of the computer is different across types, which in turn is determined by the actual design or construction of the ALU, CU, and size of the word length of the memory unit, the machine language also varies from one type of computer to another. Hence, it is important to note that after becoming proficient in the machine code of a particular

computer, the programmer may be required to learn a new machine code and would have to write all the existing programs again in case the computer system is changed.

- **Error prone** It is hard to understand and remember the various combinations of 1's and 0's representing data and instructions. This makes it difficult for a programmer to concentrate fully on the logic of the problem, thus frequently causing errors.
- **Difficult to debug and modify** Checking machine instructions to locate errors are about as tedious as writing the instructions. Further, modifying such a program is highly problematic.

Following is an example of a machine language program for adding two numbers.

Example

1. Machine Code	Comments
0011 1100	Load A register with value 7
0000 0111	
0000 0110	Load B register with 10
0000 1010	
1000 0000	A = A + B
0011 1010	Store the result into the memory location whose address is 100 (decimal)
0110 0110	
0111 0110	Halt processing

Assembly language

When symbols such as letters, digits, or special characters are employed for the operation, operand, and other parts of the instruction code, the representation is called an assembly language instruction. Such representations are known as mnemonic codes; they are used instead of binary codes. A program written with mnemonic codes forms an assembly language program. This is considered to be a *second generation language* (2GL).

Machine and assembly languages are referred to as low-level languages since the coding for a problem is at the individual instruction level. Each computer has its own assembly language that is dependent upon the internal architecture of the processor.

An *assembler* is a translator that takes input in the form of the assembly language program and produces machine language code as its output. An instruction word consists of parts shown in Fig. 1.5 where,

- the Opcode (Operation Code) part indicates the operation to be performed by the instruction and
- the mode and operand parts convey the address of the data to be found or stored.

The following is an example of an assembly language program for adding two numbers X and Y and storing the result in some memory location.

Example

2. Mnemonics	Comments	Register/ Location
LD A, 7	Load register A with 7	⇒ A 7
LD B, 10	Load register B with 10	⇒ B 10
ADD A, B	A + B: Add contents of A with contents of B and store result in register A	⇒ A 17
LD (100), A	Save the result in the main memory location 100	⇒ 100 17
HALT	Halt process	

From this example program, it is clear that using mnemonics such as LD, ADD, and HALT, the readability of the program has improved significantly.

An assembly language program cannot be executed by a machine directly as it is not in a binary machine language form. An assembler is needed to translate an assembly language program into the object code, which can then be executed by the machine. The object code is the machine language code. This is illustrated in Fig. 1.6.

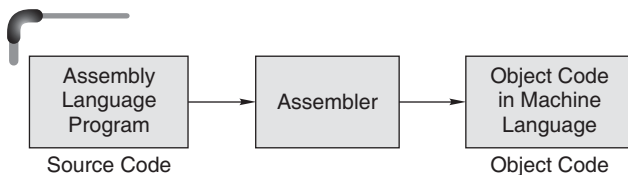


Figure 1.6 Assembler

Advantage of assembly language Writing a program in assembly language is more convenient than writing one

in machine language. Instead of binary sequence, as in machine language, a program in assembly language is written in the form of symbolic instructions. This gives the assembly language program improved readability.

Disadvantages of assembly language

- Assembly language is specific to a particular machine architecture, i.e., machine dependent. Assembly languages are designed for a specific make and model of a microprocessor. This means that assembly language programs written for one processor will not work on a different processor if it is architecturally different. That is why an assembly language program is not portable.
- Programming is difficult and time consuming.
- The programmer should know all about the logical structure of the computer.

1.2.4 High-level Languages

High-level programming languages such as COBOL, FORTRAN, and BASIC were mentioned earlier in the chapter. Such languages have instructions that are similar to human languages and have a set grammar that makes it easy for a programmer to write programs and identify and correct errors in them. To illustrate this point, a program written in BASIC, to obtain the sum of two numbers, is shown below.

Example

3. Stmt. No.	Program stmtnt	Comments
10	LET X = 7	Put 7 into X
20	LET Y = 10	Put 10 into Y
30	LET SUM = X + Y	Add values in X and Y and put in SUM.
40	PRINT SUM	Output the content in SUM.
50	END	Stop

The time and cost of creating machine and assembly language programs were quite high. This motivated the development of high-level languages.

Advantages of high-level programming languages

Readability Programs written in these languages are more readable than those written in assembly and machine languages.

Portability High-level programming languages can be run on different machines with little or no change. It is, therefore, possible to exchange software, leading to creation of program libraries.

Easy debugging Errors can be easily detected and removed.

Ease in the development of software Since the commands of these programming languages are closer to the English language, software can be developed with ease.

High-level languages are also called *third generation languages* (3GLs).

Points to Note

1. There are two kinds of programming languages --- the low-level and high level.
2. The high level programming language is easy to read, portable, allows swift development of programs and is easy to debug.
3. The low level programming language is not portable, takes more time to develop programs and debugging is difficult.

1.3 COMPILER, INTERPRETER, LOADER, AND LINKER

For executing a program written in a high-level language, it must be first translated into a form the machine can understand. This is done by a software called the *compiler*. The compiler takes the high-level language program as input and produces the machine language code as output for the machine to execute the program. This is illustrated in Fig. 1.7.

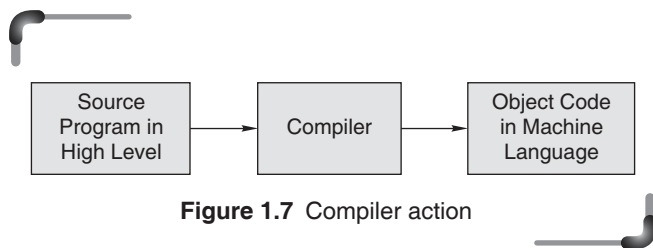


Figure 1.7 Compiler action

During the process of translation, the compiler reads the source program statement-wise and checks for syntax errors. In case of any error, the computer generates a printout of the same. This action is known as *diagnostics*.

There is another type of software that also does translation. This is called an *interpreter*.

The compiler and interpreter have different approaches to translation. Table 1.1 lists the differences between a compiler and an interpreter.

Table 1.1 Differences between a compiler and an Interpreter

Compiler	Interpreter
Scans the entire program before translating it into machine code.	Translates and executes the program line by line.
Converts the entire program to machine code and executes program only when all the syntax errors are removed.	The interpreter executes one line at a time, after checking and correcting its syntax errors and then converting it to machine code.
Slow in debugging or removal of mistakes from a program.	Good for fast debugging.
Program execution time is less.	Program execution time is more.

1.3.1 Compiling and Executing High-level Language Programs

The compiling process consists of two steps: the analysis of the source program and the synthesis of the object program in the machine language of the specified machine.

The analysis phase uses the precise description of the source programming language. A source language is described using *lexical* rules, *syntax* rules, and *semantic* rules.

Lexical rules specify the valid syntactic elements or words of the language. Syntax rules specify the way in which valid syntactic elements are combined to form the statements of the language. Syntax rules are often described using a notation known as BNF (Backus Naur Form) grammar. Semantic rules assign meanings to valid statements of the language.

The steps in the process of translating a source program in a high-level language to executable code are depicted in Fig. 1.8.

The first block is the *lexical analyzer*. It takes successive lines of a program and breaks them into individual lexical items namely, identifier, operator delimiter, etc. and attaches a type tag to each of these. Beside this, it constructs a *symbol table* for each identifier and finds the internal representation of each constant. The symbol table is used later to allocate memory to each variable.

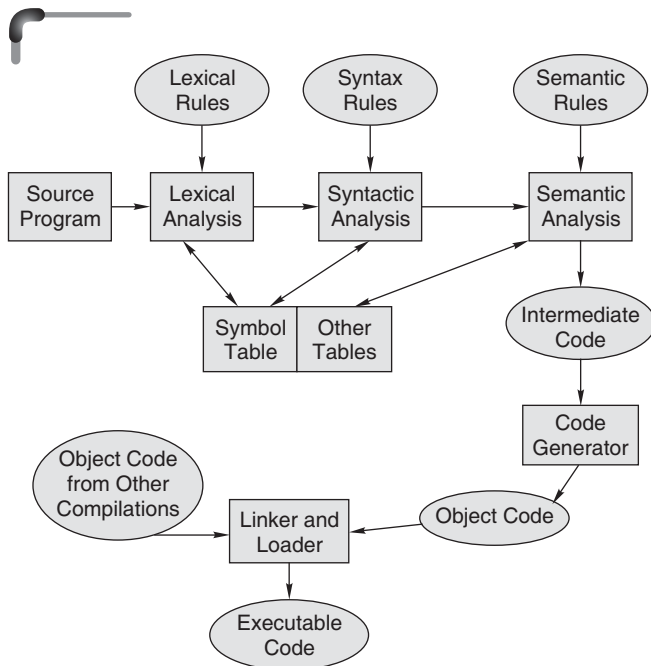


Figure 1.8 The process of compilation

The second stage of translation is called *syntax analysis* or *parsing*. In this phase, expressions, declarations, and other statements are identified by using the results of lexical analysis. Syntax analysis is done by using techniques based on formal grammar of the programming language.

In the semantic analysis phase, the syntactic units recognized by the syntax analyzer are processed. An intermediate representation of the final machine language code is produced.

The last phase of translation is code generation, when optimization to reduce the length of machine language program is carried out. The output of the code generator is a machine level language program for the specified computer. If a subprogram library is used or if some subroutines are separately translated and compiled, a final linking and loading step is needed to produce the complete machine language program in an executable form.

If subroutines were compiled separately, then the address allocation of the resulting machine language instructions would not be final. When all routines are connected and placed together in the main memory, suitable memory addresses are allocated. The linker's job is to find the correct main memory locations of the final executable program. The loader then places the executable program in memory at its correct address.

Therefore, the execution of a program written in high-level language involves the following steps:

1. *Translation* of the program resulting in the object program.
2. *Linking* of the translated program with other object programs needed for execution, thereby resulting in a binary program.
3. *Relocation* of the program to execute from the specific memory area allocated to it.
4. *Loading* of the program in the memory for the purpose of execution.

1.3.2 Linker

Linking resolves symbolic references between object programs. It makes object programs known to each other. The features of a programming language influence the linking requirements of a program. In FORTRAN/COBOL, all program units are translated separately. Hence, all subprogram calls and common variable references require linking. PASCAL procedures are typically nested inside the main program. Hence, procedure references do not require linking; they can be handled through relocation. References to built-in functions however require linking. In C, files are translated separately. Thus, only function calls that cross file boundaries and references to global data require linking. Linking makes the addresses of programs known to each other so that transfer of control from one subprogram to another or a main program takes place during execution.

Relocation

Relocation means adjustment of all address-dependent locations, such as address constant, to correspond to the allocated space, which means simple modification of the object program so that it can be loaded at an address different from the location originally specified. Relocation is more than simply moving a program from one area to another in the main memory. It refers to the adjustment of address fields. The task of relocation is to add some constant value to each relative address in the memory segment.

1.3.3 Loader

Loading means physically placing the machine instructions and data into main memory, also known as primary storage area.

A loader is a system program that accepts object programs and prepares them for execution and initiates the execution (see Fig. 1.9). The functions performed by the loader are :

- Assignment of load-time storage area to the program
- Loading of program into assigned area
- Relocation of program to execute properly from its load time storage area
- Linking of programs with one another

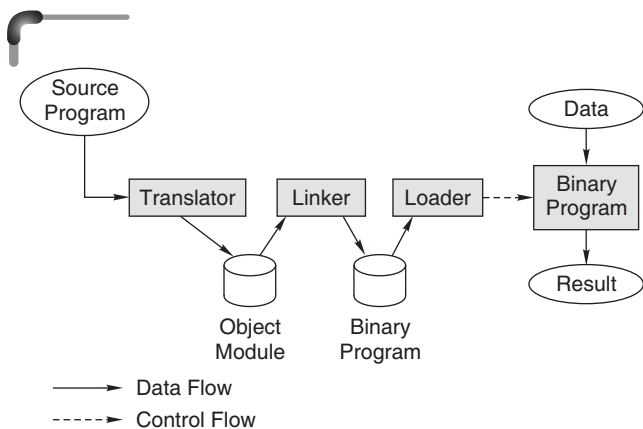


Figure 1.9 A schematic of program execution

Thus, a loader is a program that places a program's instructions and data into primary storage locations. An *absolute loader* places these items into the precise locations indicated in the machine language program. A *relocating loader* may load a program at various places in primary storage depending on the availability of primary storage area at the time of loading. A program may be relocated dynamically with the help of a *relocating register*. The base address of the program in primary storage is placed in the relocating register. The contents of the relocation register are added to each address developed by a running program. The user is able to execute the program as if it begins at location zero. At execution time, as the program runs, all address references involve the relocation register. This allows the program to reside in memory locations other than those for which it was translated to occupy.

1.3.4 Linking Loader and Linkage Editor

User programs often contain only a small portion of the instructions and data needed to solve a given problem. Large subroutine libraries are provided so that a programmer wanting to perform certain common

operations may use system-supplied routines to do so. Input/output, in particular, is normally handled by routines outside the user program. Hence, the machine language program produced by the translator must normally be combined with other machine language programs residing within the library to form a useful execution unit. This process of program combination is called linking and the software that performs this operation is variously known as a *linking loader* or a *linkage editor*. Linking is done after object code generation, prior to program execution time.

At load time, a linking loader combines whatever programs are required and loads them directly into primary storage. A linkage editor also performs the same task, but it creates a load image that it preserves on secondary storage for future reference. Whenever a program is to be executed, the load image produced by the linkage editor may be loaded immediately without the overhead of recombining program segments.

Points to Note

1. A compiler converts a high-level language program into executable machine instructions after the removal of syntax errors.
2. An interpreter executes each high-level language program one line at a time after removing its syntax error and converting it into machine instructions.
3. A linker makes the addresses of programs known to each other so that transfer of control from one subprogram to another or a main program takes place properly during execution.
4. A loader is a program that places a program's executable machine instructions and data into primary storage locations.

1.4 PROGRAM EXECUTION

The primary memory of a computer, also called the Random Access Memory, is divided into units known as words.

Depending on the computer, a word of memory may be two, four, or even eight bytes in size. Each word is associated with a unique address, which is a positive integer that helps the CPU to access the word. Addresses increase consecutively from the top of the memory to its bottom. When a program is compiled and linked, each instruction and each item of data is assigned an address. At execution time, the CPU finds instructions and data from these addresses.

The PC, or program counter, is a CPU register that holds the address of the next instruction to be executed in a program. In the beginning, the PC holds the address of the zeroth instruction of the program. The CPU fetches and then executes the instruction found at this address. The PC is meanwhile incremented to the address of the next instruction in the program. Having executed one instruction, the CPU goes back to look up the PC where it finds the address of the next instruction in the program. This instruction may not necessarily be in the next memory location. It could be at quite a different address. For example, the last statement could have been a go to statement, which unconditionally transfers control to a different point in the program; or there may have been a branch to a function subprogram. The CPU fetches the contents of the words addressed by the PC in the same amount of time, whatever their physical locations. The CPU has random access capability to any and all words of the memory, no matter what their addresses. Program execution proceeds in this way until the CPU has processed the last instruction.

Points to Note

1. When a program is compiled and linked, each instruction and each item of data is assigned an address.
2. During program execution, the CPU finds instructions and data from the assigned addresses.

1.5 FOURTH GENERATION LANGUAGES

The Fourth Generation Language is a non-procedural language that allows the user to simply specify what the output should be without describing how data should be processed to produce the result. Fourth generation programming languages are not as clearly defined as are the other earlier generation languages. Most people feel that a fourth generation language, commonly referred to as 4GL, is a high-level language that requires significantly fewer instructions to accomplish a particular task than does a third generation language. Thus, a programmer should be able to write a program faster in 4GL than in a third generation language.

Most third generation languages are procedural languages. That is, the programmer must specify the steps of the procedure the computer has to follow in a program. By contrast, most fourth generation languages are non-procedural languages. The programmer does not have

to give the details of the procedure in the program, but specify, instead, what is wanted. For example, assume that a programmer needs to display some data on the screen, such as the address of a particular employee, say MANAS, from the EMP file. In a procedural language, the programmer would have to write a series of instructions using the following steps:

Step 1: Get a record from the EMP file.

Step 2: If this is the record for MANAS, display the address.

Step 3: If this is not the record for MANAS, go to step 1, until end-of-file.

In a non-procedural language (4GL), however, the programmer would write a single instruction that says:

Get the address of MANAS from EMP file.

Major fourth generation languages are used to get information from files and databases, as in the previous example, and to display or print the information. These fourth generation languages contain a query language, which is used to answer queries or questions with data from a database. The following example shows a query in a common query language, SQL.

```
SELECT ADDRESS FROM EMP WHERE NAME = 'MANAS'
```

End user-oriented 4GLs are designed for applications that process low data volumes. These 4GLs run on mainframe computers and may be employed either by information users or by the programmers. This type of 4GL may have its own internal database management software that in turn interacts with the organization's DBMS package. People who are not professional programmers use these products to query databases, develop their own custom-made applications, and generate their own reports with minimum amount of training. For example, ORACLE offers a number of tools suitable for the end user.

Some fourth generation languages are used to produce complex printed reports. These languages contain certain types of programs called generators. With a report generator, the programmer specifies the headings, detailed data, and totals needed in a report. Thus, the report generator produces the required report using data from a file. Other fourth generation languages are used to design screens for data input and output and for menus. These languages

contain certain types of programs called screen painters. The programmer designs the screen to look as desired and, therefore, it can be said that the programmer paints the screen using the screen painter program. Fourth generation languages are mostly machine independent. Usually they can be used on more than one type of computer. They are mostly used for office automation or business applications, and not for scientific programs. Some fourth generation languages are designed to be easily learnt and employed by end users.

Advantages of 4GLs

- Programming productivity is increased. One line of a 4GL code is equivalent to several lines of a 3GL code.
- System development is faster.
- Program maintenance is easier.
- End users can often develop their own applications.
- Programs developed in 4GLs are more portable than those developed in other generation languages.
- Documentation is of improved order because most 4GLs are self-documenting.

The differences between third generation languages and fourth generated languages are shown in Table 1.2.

Table 1.2 3GL vs 4GL

3GL	4GL
Meant for use by professional programmers	May be used by non-professional programmers as well as by professional programmers.
Requires specifications of how to perform a task	Requires specifications of what task to perform.
All alternatives must be specified	System determines how to perform the task.
Execution time is less	Default alternatives are built-in. User need not specify these alternatives.
Requires large number of procedural instructions Code may be difficult to read, understand, and maintain by the user	Requires fewer instructions.
Typically, file oriented	Difficult to debug

1.6 FIFTH GENERATION LANGUAGES

Natural languages represent the next step in the development of programming languages belonging to fifth generation languages. Natural language is similar to query language, with one difference: it eliminates the need for the user or programmer to learn a specific vocabulary, grammar, or syntax. The text of a natural-language statement resembles human speech closely. In fact, one could word a statement in several ways, perhaps even misspelling some words or changing the order of the words, and get the same result. Natural language takes the user one step further away from having to deal directly and in detail with computer hardware and software. These languages are also designed to make the computer smarter—that is, to simulate the human learning process. Natural languages already available for microcomputers include CLOUT, Q & A, and SAVY RETRIEVER (for use with databases) and HAL(Human Access Language) for use with LOTUS.

Points to Note

1. Third generation programming language specifies how to perform a task using a large number of procedural instructions and is file oriented.
2. Fourth generation programming language specifies what task has to be performed using fewer instructions and is database oriented.
3. Fifth generation programming language resembles human speech and eliminates the need for the user or programmer to learn a specific vocabulary, grammar, or syntax.

1.7 CLASSIFICATION OF PROGRAMMING LANGUAGES

1.7.1 Procedural Languages

Algorithmic languages These are high-level languages designed for forming convenient expression of procedures, used in the solution of a wide class of problems. In this language, the programmer must specify the steps the computer has to follow while executing a program. Some of languages that fall in the category are C, COBOL, and FORTRAN.

Object-oriented language The basic philosophy of object-oriented programming is to deal with objects rather than functions or subroutines as in strictly algorithmic languages.

Objects are self-contained modules that contain data as well as the functions needed to manipulate the data within the same module. In a conventional programming language, data and subroutines or functions are separate. In object-oriented programming, subroutines as well as data are locally defined in objects. The difference affects the way a programmer goes about writing a program as well as how information is represented and activated in the computer. The most important object-oriented programming features are

- abstraction
- encapsulation and data hiding
- polymorphism
- inheritance
- reusable code

C++, JAVA, SMALLTALK, etc. are examples of object-oriented languages.

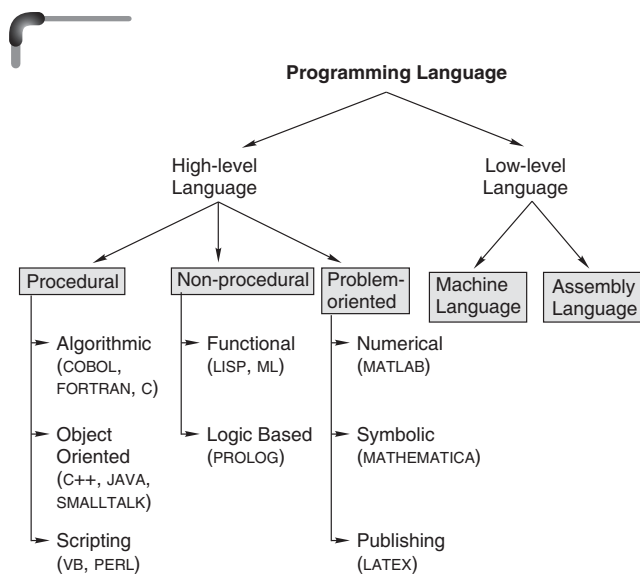


Fig. 1.10 Programming language classification

Scripting languages These languages assume that a collection of useful programs, each performing a task, already exists. It has facilities to combine these components to perform a complex task. A scripting language may thus be thought of as a glue language, which sticks a variety of components together. One of the earliest scripting languages is the UNIX shell. Now there are several scripting languages such as VB script and Perl.

1.7.2 Problem-oriented Languages

These are high-level languages designed for developing a convenient expression of a given class of problems.

1.7.3 Non-procedural Languages

Functional (applicative) languages These functional languages solve a problem by applying a set of functions to the initial variables in specific ways to get the answer. The functional programming style relies on the idea of function application rather than on the notion of variables and assignments. A program written in a functional language consists of function calls together with arguments to functions. LISP, ML, etc. are examples of functional languages.

Logic-based programming language A logic program is expressed as a set of atomic sentences, known as fact, and horn clauses, such as if-then rules. A query is then posed. The execution of the program now begins and the system tries to find out if the answer to the query is true or false for the given facts and rules. Such languages include PROLOG.

Points to Note

1. Programming languages can be categorized as high-level or low-level.
2. High-level languages are classified as procedural, non-procedural and problem-oriented languages. Programs in high-level languages are easy to prepare and debug. Such languages are not machine oriented.
3. Low-level languages are machine oriented languages.

1.8 STRUCTURED PROGRAMMING CONCEPT

In 1968, computer scientist Edsger Dijkstra of Netherlands published a letter to the editor in the journal of the Association of Computing Machinery with the title 'Go To statement considered harmful'. goto is a command available in most programming languages to transfer a control to a particular statement. For three decades, Dijkstra had been crusading for a better way of programming—a systematic way to organize programs—called structured programming.

Structured programming has been called a revolution in programming and is considered as one of the most important advancements in software in the past two decades. Both academic and industrial professionals are inclined towards the philosophy and techniques of structured programming. Today, it can be safely said that

virtually all software developers acknowledge the merits of the structured programming approach and use it in software development.

There is no standard definition of structured programs available but it is often thought to be programming without the use of a goto statement. Indeed, structured programming does discourage the frequent use of goto but there is more to it than that.

Structured programming is:

- concerned with improving the programming process through better organization of programs and better programming notation to facilitate correct and clear description of data and control structure.
- concerned with improved programming languages and organized programming techniques which should be understandable and therefore, more easily modifiable and suitable for documentation.
- more economical to run because good organization and notation make it easier for an optimizing compiler to understand the program logic.
- more correct and therefore more easily debugged, because general correctness theorems dealing with structures can be applied to proving the correctness of programs.

Structured programming can be defined as a

- top-down analysis for program solving
- modularization for program structure and organization
- structured code for individual modules

1.8.1 Top-Down Analysis

A program is a collection of instructions in a particular language that is prepared to solve a specific problem. For larger programs, developing a solution can be very complicated. From where should it start? Where should it terminate? Top-down analysis is a method of problem solving and problem analysis. The essential idea is to subdivide a large problem into several smaller tasks or parts for ease of analysis.

Top-down analysis, therefore, simplifies or reduces the complexity of the process of problem solving. It is not limited by the type of program. Top-down analysis is a general method for attending to any problem. It provides a strategy that has to be followed for solving all problems.

There are two essential ideas in top-down analysis:

- subdivision of a problem
- hierarchy of tasks

Subdivision of a problem means breaking a big problem into two or more smaller problems. Therefore, to solve the big problem, first these smaller problems have to be solved.

Top-down analysis does not simply divide a problem into two or more smaller problems.

It goes further than that. Each of these smaller problems is further subdivided. This process continues downwards, creating a hierarchy of tasks, from one level to the next, until no further break up is possible.

The four basic steps to top-down analysis are as follows:

Step 1: Define the complete scope of the problem to determine the basic requirement for its solution. Three factors must be considered in the definition of a programming problem.

- Input: What data is required to be processed by the program?
- Process: What must be done with the input data? What type of processing is required?
- Output: What information should the program produce? In what form should it be presented?

Step 2: Based on the definition of the problem, divide the problem into two or more separate parts.

Step 3: Carefully define the scope of each of these separate tasks and subdivide them further, if necessary, into two or more smaller tasks.

Step 4: Repeat step 3. Every step at the lowest level describes a simple task, which cannot be broken further.

1.8.2 Modular Programming

Modular programming is a program that is divided into logically independent smaller sections, which can be written separately. These sections, being separate and independent units, are called modules.

- A module consists of a series of program instructions or statements in some programming language.
- A module is clearly terminated by some special markers required by the syntax of the language. For example, a BASIC language subroutine is terminated by the return statement.
- A module as a whole has a unique name.
- A module has only one entry point to which control is transferred from the outside and only one exit point from which control is returned to the calling module.

The following are some advantages of modular programming.

- Complex programs may be divided into simpler and more manageable elements.
- Simultaneous coding of different modules by several programmers is possible.
- A library of modules may be created, and these modules may be used in other programs as and when needed.
- The location of program errors may be traced to a particular module; thus, debugging and maintenance may be simplified.

1.8.3 Structured Code

After the top-down analysis and design of the modular structure, the third and final phase of structured programming involves the use of structured code. Structured programming is a method of coding, i.e., writing a program that produces a well-organized module.

A high-level language supports several control statements, also called structured control statements or structured code, to produce a well-organized structured module. These control statements represent conditional and repetitive type of executions. Each programming language has different syntax for these statements.

In C, the `if` and `case` statements are examples of conditional execution whereas `for`, `while`, and `do...while` statements represent repetitive execution. In BASIC, `for-next` and `while-wend` are examples of repetitive execution. Let us consider the `goto` statement of BASIC, which is a simple but not a structured control statement. The `goto` statement can break the normal flow of the program and transfer control to any arbitrary point in a program. A module that does not have a normal flow control is unorganized and unreadable.

The following example is a demonstration of a program using several `goto` statements. Note that at line numbers 20, 60, and 80, the normal flow control is broken. For example, from line number 60, control goes back to line 40 instead of line 70 in case value of $(R - G)$ is less than 0.001.

```
10 INPUT X
20 IF X < 0 THEN GOTO 90
30 G = X/2
40 R = X/G
50 G = (R + G)/2
60 IF ABS(R - G) < 0.001 THEN GOTO 40
70 PRINT G
```

```
80 GOTO 100
90 PRINT "INVALID INPUT"
100 END
```

The structured version of this program using `while-wend` statement is given below.

```
INPUT X
IF X > 0
  THEN
    G = X/2
    R = X/G
    WHILE ABS (R - G) < 0.001
      R = X/G
      G = (R + G)/2
    WEND
    PRINT G
  ELSE
    PRINT "INVALID INPUT"
  END
```

Now if there is no normal break of control flow, `gotos` are inevitable in unstructured languages but they can be and should be always avoided while using structured programs except in unavoidable situations.

1.8.4 The Process of Programming

The job of a programmer is not just writing program instructions. The programmer does several other additional jobs to create a working program. There are some logical and sequential job steps which the programmer has to follow to make the program operational.

These are as follows:

1. Understand the problem to be solved
2. Think and design the solution logic
3. Write the program in the chosen programming language
4. Translate the program to machine code
5. Test the program with sample data
6. Put the program into operation

The first job of the programmer is to understand the problem. To do that the requirements of the problem should be clearly defined. And for this, the programmer may have to interact with the user to know the needs of the user. Thus this phase of the job determines the 'what to' of the task.

The next job is to develop the logic of solving the problem. Different solution logics are designed and the order in which these are to be used in the program are defined. Hence, this phase of the job specifies the 'how to' of the task.

Once the logics are developed, the third phase of the job is to write the program using a chosen programming language. The rules of the programming language have to be observed while writing the program instructions.

The computer recognizes and works with 1's and 0's. Hence program instructions have to be converted to 1's and 0's for the computer to execute it. Thus, after the program is written, it is translated to the machine code, which is in 1's and 0's with the help of a translating program.

Now, the program is tested with dummy data. Errors in the programming logic are detected during this phase and are removed by making necessary changes in either the logic or the program instructions.

The last phase is to make the program operational. This means, the program is put to actual use. Errors occurring in this phase are rectified to finally make the program work to the user's satisfaction.

Points to Note

1. Structured programming involves top-down analysis for program solving, modularization of program structure and organizing structured code for individual module.
2. Top-down analysis breaks the whole problem into smaller logical tasks and defines the hierarchical link between the tasks.
3. Modularization of program structure means making the small logical tasks into independent program modules that carries out the desired tasks.
4. Structured coding is structured programming which consists of writing a program that produces a well-organized module.

1.9 ALGORITHMS

1.9.1 What is an Algorithm?

Computer scientist Niklaus Wirth stated that

Program = Algorithms + Data

An algorithm is a part of the plan for the computer program. In fact, an algorithm is 'an effective procedure for solving a problem in a finite number of steps'.

It is effective, which means that an answer is found and it has a finite number of steps. A well-designed algorithm will always provide an answer; it may not be the desired answer but there will be an answer. It may be that the answer is that there is no answer. A well-designed algorithm is also guaranteed to terminate.

1.9.2 Different Ways of Stating Algorithms

Algorithms may be represented in various ways. There are four ways of stating algorithms.

These are as follows:

- Step-form
- Pseudo-code
- Flowchart
- Nassi-Schneiderman

In the step form representation, the procedure of solving a problem is stated with written statements. Each statement solves a part of the problem and these together complete the solution. The step-form uses just normal language to define each procedure. Every statement, that defines an action, is logically related to the preceding statement. This algorithm has been discussed in the following section with the help of an example.

The pseudo-code is a written form representation of the algorithm. However it differs from the step form as it uses a restricted vocabulary to define its action of solving the problem. One problem with human language is that it can seem to be imprecise. But the pseudo-code, which is in human language, tends toward more precision by using a limited vocabulary.

Flowchart and Nassi-Schneiderman are graphically oriented representation forms. They use symbols and language to represent sequence, decision, and repetition actions. Only the flowchart method of representing the problem solution has been explained with several examples. The Nassi-Schneiderman technique is beyond the scope of this book.

Points to Note

1. An algorithm is an effective procedure for solving a problem in a finite number of steps.
2. A program is composed of algorithm and data.
3. The four common ways of representing an algorithm are the Step-form, Pseudo-code, Flowchart and Nassi-Schneiderman.

1.9.3 Key Features of an Algorithm and the Step-form

Here is an example of an algorithm, for making a pot of tea.

1. If the kettle does not contain water, then fill the kettle.
2. Plug the kettle into the power point and switch it on.
3. If the teapot is not empty, then empty the teapot.

4. Place tea leaves in the teapot.
5. If the water in the kettle is not boiling, then go to step 5.
6. Switch off the kettle.
7. Pour water from the kettle into the teapot.

It can be seen that the algorithm has a number of steps and that some steps (steps 1, 3, and 5) involve decision making and one step (step 5 in this case) involves repetition, in this case the process of waiting for the kettle to boil.

From this example, it is evident that algorithms show these three features:

- Sequence (also known as process)
- Decision (also known as selection)
- Repetition (also known as iteration or looping)

Therefore, an algorithm can be stated using three basic constructs: sequence, decision, and repetition.

Sequence

Sequence means that each step or process in the algorithm is executed in the specified order. In the above example, each process must be in the proper place otherwise the algorithm will fail.

The decision constructs—*if ... then, if ... then ... else ...*

In algorithms the outcome of a decision is either true or false; there is no state in between.

The outcome of the decision is based on some condition that can only result in a true or false value. For example,

```
if today is Friday then collect pay
```

is a decision and the decision takes the general form:

```
if proposition then process
```

A proposition, in this sense, is a statement, which can only be true or false. It is either true that 'today is Friday' or it is false that 'today is not Friday'. It can not be both true and false. If the proposition is true, then the process or procedure that follows the then is executed. The decision can also be stated as:

```
if proposition
  then process1
  else process2
```

This is the *if ... then ... else ...* form of the decision. This means that if the proposition is true then execute process1, else, or otherwise, execute process2.

The first form of the decision if proposition then process has a null else, that is, there is no else.

The repetition constructs—*repeat and while*

Repetition can be implemented using constructs like the repeat loop, while loop, and *if.. then .. goto .. loop*.

The Repeat loop is used to iterate or repeat a process or sequence of processes until some condition becomes true.

It has the general form:

```
Repeat
  Process1
  Process2
  .....
  .....
  ProcessN
  Until proposition
```

Here is an example.

```
Repeat
  Fill water in kettle
  Until kettle is full
```

The process is 'Fill water in kettle,' the proposition is 'kettle is full'.

The Repeat loop does some processing before testing the state of the proposition.

What happens though if in the above example the kettle is already full? If the kettle is already full at the start of the Repeat loop, then filling more water will lead to an overflow.

This is a drawback of the Repeat construct.

In such a case the *while* loop is more appropriate. The above example with the *while* loop is shown as follows:

```
while kettle is not full
  fill water in kettle
```

Since the decision about the kettle being full or not is made before filling water, the possibility of an overflow is eliminated. The while loop finds out whether some condition is true before repeating a process or a sequence of processes.

If the condition is false, the process or the sequence of processes is not executed. The general form of *while* loop is:

```
while proposition
  begin
  Process 1
```

```

Process 2
.....
.....
Process N
end

```

The `if .. then goto ..` is also used to repeat a process or a sequence of processes until the given proposition is false. In the kettle example, this construct would be implemented as follows:

1. Fill some water in kettle
2. `if kettle not full then goto 1`

So long as the proposition ‘kettle not full’ is true the process, ‘fill some water in kettle’ is repeated. The general form of `if .. then goto ..` is:

```

Process1
Process2
.....
.....
ProcessN
if proposition then goto Process1

```

Termination

The definition of algorithm cannot be restricted to procedures that eventually finish. Algorithms might also include procedures that could run forever without stopping. Such a procedure has been called a computational method by Knuth or calculation procedure or algorithm by Kleene. However, Kleene notes that such a method must eventually exhibit ‘some object.’ Minsky (1967) makes the observation that, if an algorithm has not terminated, then how can the following question be answered: “Will it terminate with the correct answer?” Thus the answer is: undecidable. It can never be known, nor can the designer do an analysis beforehand to find it out. The analysis of algorithms for their likelihood of termination is called termination analysis.

Correctness

The prepared algorithm needs to be verified for its correctness. Correctness means how easily its logic can be argued to meet the algorithm’s primary goal. This requires the algorithm to be made in such a way that all the elements in it are traceable to the requirements.

Correctness requires that all the components like the data structures, modules, external interfaces, and module interconnections are completely specified.

In other words, correctness is the degree to which an algorithm performs its specified function. The most common measure of correctness is defects per Kilo Lines of Code (KLOC) that implements the algorithm, where defect is defined as the verified lack of conformance to requirements.

Points to Note

1. The key features of an algorithm are sequence, selection and repetition.
2. The stepwise form has sequence, selection and repetition constructs.
3. Termination means the action of closing. A well-designed algorithm has a termination.
4. Correctness of algorithm means how easily its logic can be argued to meet the algorithm’s primary goal.

1.9.4 What are Variables?

So long, the elements of algorithm have been discussed. But a program comprises of algorithm and data. Therefore, it is now necessary to understand the concept of data. It is known that data is a symbolic representation of value and that programs set the context that gives data a proper meaning. In programs, data is transformed into information. The question is, how is data represented in programs?

Almost every algorithm contains data and usually the data is ‘contained’ in what is called a variable. The variable is a container for a value that may vary during the execution of the program. For example, in the tea-making algorithm, the level of water in the kettle is a variable, the temperature of the water is a variable, and the quantity of tea leaves is also a variable.

Each variable in a program is given a name, for example,

- Water_Level
- Water_Temperature
- Tea_Leaves_Quantity

and at any given time the value, which is represented by `Water_Level`, for instance, may be different to its value at some other time. The statement

```
if the kettle does not contain water then fill the kettle
```

could also be written as

```
if Water_Level is 0 then fill the kettle
```

or

```
if Water_Level = 0 then fill the kettle
```

At some point `Water_Level` will be the maximum value, whatever that is, and the kettle will be full.

Variables and data types

The data used in algorithms can be of different types. The simplest types of data that an algorithm might use are

- numeric data, e.g., 12, 11.45, 901, etc.
- alphabetic or character data such as 'A', 'Z', or 'This is alphabetic'
- logical data, that is, propositions with true/false values

Naming of variables

One should always try to choose meaningful names for variables in algorithms to improve the readability of the algorithm or program. This is particularly important in large and complex programs.

In the tea-making algorithm, plain English was used. It has been shown how variable names may be used for some of the algorithm variables. In Table 1.3, the right-hand column contains variable names which are shorter than the original and do not hide the meaning of the original phrase. Underscores have been given to indicate that the words belong together and represent a variable.

Table 1.3 Algorithm using variable names

Algorithm in Plain English	Algorithm using Variable Names
1. If the kettle does not contain water, then fill the kettle.	1. If kettle_empty then fill the kettle.
2. Plug the kettle into the power point and switch it on.	2. Plug the kettle into the power point and switch it on.
3. If the teapot is not empty, then empty the teapot.	3. If teapot_not_empty then empty the teapot.
4. Place tea leaves in the teapot.	4. Place tea leaves in the teapot.
5. If the water in the kettle is not boiling then go to step 5.	5. If water_not_boiling then go to step 5.
6. Switch off the kettle.	6. Switch off the kettle.
7. Pour water from the kettle into the teapot.	7. Pour water from the kettle into the teapot.

There are no hard and fast rules about how variables should be named but there are many conventions. It is a good idea to adopt a conventional way of naming variables.

The algorithms and programs can benefit from using naming conventions for processes too.

Points to Note

1. Data is a symbolic representation of value.
2. A variable, which has a name, is a container for a value that may vary during the execution of the program.

1.9.5 Subroutines

A simple program is a combination of statements that are implemented in a sequential order. A statement block is a group of statements. Such a program is shown in Fig. 1.11(i). There might be a specific block of statement, which is also known as a procedure, that is run several times at different points in the implementation sequence of the larger program. This is shown in Fig. 1.11(ii). Here, this specific block of statement is named "procedure X". In this example program, the "procedure X" is written twice in this example. This enhances the size of the program. Since this particular procedure is required to be run at two specific points in the implementation sequence of the larger program, it may be treated as a separate entity and not included in the main program. In fact, this procedure may be called whenever required as shown in Fig. 1.11(iii). Such a procedure is known as a subroutine.

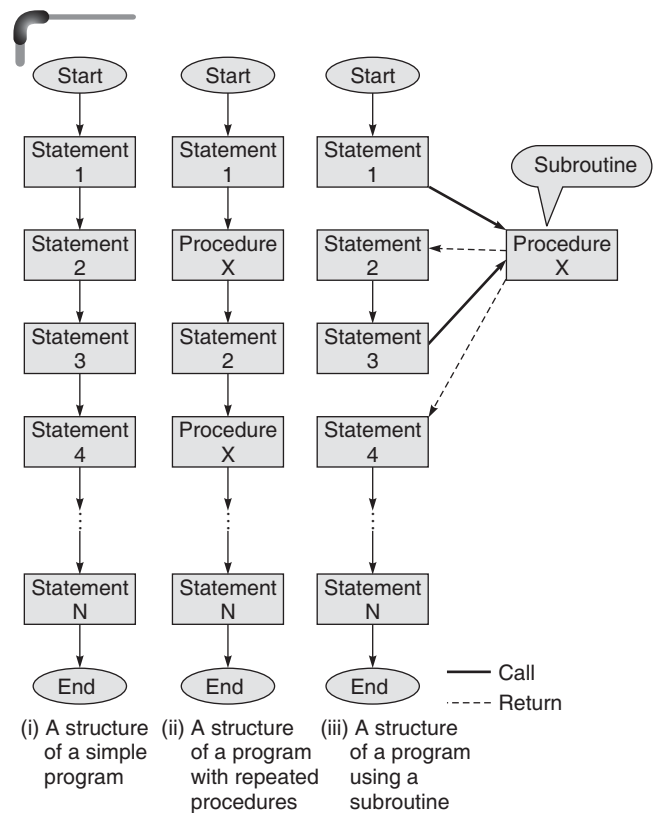


Figure 1.11 Program structures

Therefore, a subroutine, also known as procedure, method or function, is a portion of instruction that is invoked from within a larger program to perform a specific task. At the same time the subroutine is relatively independent of the remaining statements of the larger program. The subroutine behaves in much the same way as a program that is used as one step in a larger program. A subroutine is often written so that it can be started (“called”) several times and/or from several places during a single execution of the program, including from other subroutines, and then branch back (*return*) to the next instruction after the “call”, once the subroutine’s task is done. Thus, such subroutines are invoked with a CALL statement with or without passing of parameters from the calling program. The subroutine works on the parameters if given to it, otherwise it works out the results and gives out the result by itself and returns to the calling program or pass the results to the calling program before returning to it.

The technique of writing subroutine has some distinct advantages. The subroutine reduces duplication of block of statements within a program, enables reuse of the block of statements that forms the subroutine across multiple programs, decomposes a complex task into simpler steps, divides a large programming task among various programmers or various stages of a project and hides implementation details from users.

However, there are some disadvantages in using subroutines. The starting or invocation of a subroutine requires some computational overhead in the call mechanism itself. The subroutine requires some well defined housekeeping techniques at its entry and exit from it.

Points to Note

1. A subroutine is a logical collection of instructions that is invoked from within a larger program to perform a specific task.
2. The subroutine is relatively independent of the remaining statements of the program that invokes it.
3. A subroutine can be invoked several times from several places during a single execution of the invoking program.
4. After completing the specific task, a subroutine returns to the point of invocation in the larger program.

Some examples on developing algorithms using step-form

For illustrating the step-form the following conventions are assumed:

1. Each algorithm will be logically enclosed by two statements START and STOP.
2. To accept data from user, the INPUT or READ statements are to be used.
3. To display any user message or the content in a variable, PRINT statement will be used. Note that the message will be enclosed within quotes.
4. There are several steps in an algorithm. Each step results in an action. The steps are to be acted upon sequentially in the order they are arranged or directed.
4. The arithmetic operators that will be used in the expressions are

- (i) ‘←’Assignment (the left-hand side of ‘←’ should always be a single variable)

Example: The expression $x \leftarrow 6$ means that a value 6 is assigned to the variable x. In terms of memory storage, it means a value of 6 is stored at a location in memory which is allocated to the variable x.

- (ii) ‘+’ Addition

Example: The expression $z \leftarrow x + y$ means the value contained in variable x and the value contained in variable y is added and the resulting value obtained is assigned to the variable z.

- (iii) ‘-’ Subtraction

Example: The expression $z \leftarrow x - y$ means the value contained in variable y is subtracted from the value contained in variable x and the resulting value obtained is assigned to the variable z

- (iv) ‘*’ Multiplication

Example: Consider the following expressions written in sequence:

$$\begin{aligned}x &\leftarrow 5 \\y &\leftarrow 6 \\z &\leftarrow x * y\end{aligned}$$

The result of the multiplication between x and y is 30. This value is therefore assigned to z.

- (v) ‘/’ Division

Example: The following expressions written in sequence illustrates the meaning of the division operator :

$$\begin{aligned}x &\leftarrow 10 \\y &\leftarrow 6 \\z &\leftarrow x/y\end{aligned}$$

The quotient of the division between x and y is 1 and the remainder is 4. When such an operator is used the quotient is taken as the result whereas the remainder is rejected. So here the result obtained from the expression x/y is 1 and this is assigned to z .

5. In propositions, the commonly used relational operators will include

(i) ' $>$ ' Greater than

Example: The expression $x > y$ means if the value contained in x is larger than that in y then the outcome of the expression is true, which will be taken as 1. Otherwise, if the outcome is false then it would be taken as 0.

(ii) ' $<=$ ' Less than or equal to

Example: The expression $x <= y$ implies that if the value held in x is either less than or equal to the value held in y then the outcome of the expression is true and so it will be taken as 1.

But if the outcome of the relational expression is false then it is taken as 0.

(iii) ' $<$ ' Less than

Example: Here the expression $x < y$ implies that if the value held in x is less than that held in y then the relational expression is true, which is taken as 1, otherwise the expression is false and hence will be taken as 0.

(iv) ' $=$ ' Equality

Example: The expression $x = y$ means that if the value in x and that in y are same then this relational expression is true and hence the outcome is 1 otherwise the outcome is false or 0.

(v) ' $>=$ ' Greater than or equal to

Example: The expression $x >= y$ implies that if the value in x is larger or equal to that in y then the outcome of the expression is true or 1, otherwise it is false or 0.

(vi) ' $!=$ ' Non-equality

Example: The expression $x != y$ means that if the value contained in x is not equal to the value contained in y then the outcome of the expression is true or 1, otherwise it is false or 0.

Note: The 'equal to ($=$)' operator is used both for assignment as well as equality specification. When used in proposition, it specifies equality otherwise assignment. To differentiate 'assignment' from

'equality' left arrow (\leftarrow) may be used. For example, $a \leftarrow b$ is an assignment but $a = b$ is a proposition for checking the equality.

6. The most commonly used logical operators will be AND, OR and NOT. These operators are used to specify multiple test conditions forming composite proposition. These are

(i) 'AND' Conjunction

The outcome of an expression is true or 1 when both the propositions AND-ed are true otherwise it is false or 0.

Example: Consider the expressions

$x \leftarrow 2$

$y \leftarrow 1$

$x = 2$ AND $y = 0$

In the above expression the proposition ' $x = 2$ ' is true because the value in x is 2. Similarly, the proposition ' $y = 0$ ' is untrue as y holds 1 and therefore this proposition is false or 0. Thus, the above expression may be represented as 'true' AND 'false' the outcome for which is false or 0.

(ii) 'OR' Disjunction

The outcome of an expression is true or 1 when anyone of the propositions OR-ed is true otherwise it is false or 0.

Example: Consider the expressions

$x \leftarrow 2$

$y \leftarrow 1$

$x = 2$ OR $y = 0$

Here, the proposition ' $x = 2$ ' is true since x holds 2 while the proposition ' $y = 0$ ' is untrue or false. Hence the third expression may be represented as 'true' OR 'false' the outcome for which is true or 1.

(iii) 'NOT' Negation

If outcome of a proposition is 'true', it becomes 'false' when negated or NOT-ed.

Example: Consider the expression

$x \leftarrow 2$

NOT $x = 2$

The proposition ' $x = 2$ ' is 'true' as x contains the value 2. But the second expression negates this by the logical operator NOT which gives an outcome 'false'.

Examples

4. Write the algorithm for finding the sum of any two numbers.

Solution Let the two numbers be A and B and let their sum be equal to C . Then, the desired algorithm is given as follows:

1. START
2. PRINT "ENTER TWO NUMBERS"
3. INPUT A, B
4. $C \leftarrow A + B$
5. PRINT C
6. STOP

Add values assigned to A and B and assign this value to C

Explanation The first step is the starting point of the algorithm. The next step requests the programmer to enter the two numbers that have to be added. Step 3 takes in the two numbers given by the programmer and keeps them in variables A and B . The fourth step adds the two numbers and assigns the resulting value to the variable C . The fifth step prints the result stored in C on the output device. The sixth step terminates the procedure.

5. Write the algorithm for determining the remainder of a division operation where the dividend and divisor are both integers.

Solution Let N and D be the dividend and divisor, respectively. Assume Q to be the quotient, which is an integer, and R to be the remainder. The algorithm for the given problem is as follows.

1. START
2. PRINT "ENTER DIVIDEND"
3. INPUT N
4. PRINT "ENTER DIVISOR"
5. INPUT D
6. $Q \leftarrow N/D$ (Integer division)
7. $R \leftarrow N - Q * D$
8. PRINT R
9. STOP

Only integer value is obtained and remainder ignored

Explanation The first step indicates the starting point of the algorithm. The next step asks the programmer to enter the dividend value. The third step keeps the dividend value in the variable N . Step 4 asks for the divisor value to be entered. This is kept in the variable D . In step 6, the value in N is divided by that in D . Since both the numbers are integers, the result is an integer. This value is assigned to Q . Any remainder in this step is ignored. In step 7, the remainder is computed by subtracting the product of the integer quotient and the integer divisor from integer dividend N . The computed value of the remainder is an integer here and obviously less than the divisor. The remainder

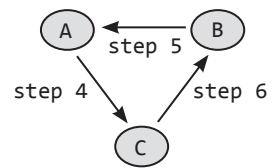
value is assigned to the variable R . This value is printed on an output device in step 8. Step 9 terminates the algorithm.

6. Construct the algorithm for interchanging the numeric values of two variables.

Solution Let the two variables be A and B . Consider C to be a third variable that is used to store the value of one of the variables during the process of interchanging the values.

The algorithm for the given problem is as follows.

1. START
2. PRINT "ENTER THE VALUE OF A & B "
3. INPUT A, B
4. $C \leftarrow A$
5. $A \leftarrow B$
6. $B \leftarrow C$
7. PRINT A, B
8. END



Explanation Like the previous examples, the first step indicates the starting point of the algorithm. The second step is an output message asking for the two values to be entered. Step 3 puts these values into the variables A and B . Now, the value in variable A is copied to variable C in step 4. In fact the value in A is saved in C . In step 5 the value in variable B is assigned to variable A . This means a copy of the value in B is put in A . Next, in step 6 the value in C , saved in it in the earlier step 4 is copied into B . In step 7 the values in A and B are printed on an output device. Step 8 terminates the procedure.

7. Write an algorithm that compares two numbers and prints either the message identifying the greater number or the message stating that both numbers are equal.

Solution This example demonstrates how the process of selection or decision making is implemented in an algorithm using the step-form. Here, two variables, A and B , are assumed to represent the two numbers that are being compared. The algorithm for this problem is given as follows.

1. START
2. PRINT "ENTER TWO NUMBERS"
3. INPUT A, B
4. IF $A > B$ THEN
 PRINT "A IS GREATER THAN B"
5. IF $B > A$ THEN
 PRINT "B IS GREATER THAN A"
6. IF $A = B$ THEN
 PRINT "BOTH ARE EQUAL"
7. STOP

Explanation The first step indicates the starting point of the algorithm. The next step prints a message asking for the entry of the two numbers. In step 3 the numbers entered are kept in the variables A and B . In steps 4, 5 and 6, the values in A , B and C compared with the IF ...THEN construct. The relevant message is printed whenever the proposition between IF and THEN is found to agree otherwise the next step is acted upon. But in any case one of the message would be printed because at least one of the propositions would be true. Step 7 terminates the procedure.

8. Write an algorithm to check whether a number given by the user is odd or even.

Solution Let the number to be checked be represented by N . The number N is divided by 2 to give an integer quotient, denoted by Q . If the remainder, designated as R , is zero, N is even; otherwise N is odd. This logic has been applied in the following algorithm.

1. START
2. PRINT "ENTER THE NUMBER"
3. INPUT N
4. $Q \leftarrow N/2$ (Integer division)
5. $R \leftarrow N - Q * 2$
6. IF $R = 0$ THEN
 PRINT "N IS EVEN"
7. IF $R \neq 0$ THEN
 PRINT "N IS ODD"
8. STOP

Explanation The primary aim here is to find out whether the remainder after the division of the number with 2 is zero or not. If the number is even the remainder after the division will be zero. If it is odd, the remainder after the division will not be zero. So by testing the remainder it is possible to determine whether the number is even or odd.

The first step indicates the starting point of the algorithm while the next prints a message asking for the entry of the number. In step 3, the number is kept in the variable N . N is divided by 2 in step 4. This operation being an integer division, the result is an integer. This result is assigned to Q . Any remainder that occurs is ignored. Now in step 5, the result Q is multiplied by 2 which obviously produces an integer that is either less than the value in N or equal to it. Hence in step 5 the difference between N and $Q * 2$ gives the remainder. This remainder value is then checked in step 6 and step 7 to either print out that it is either even or odd respectively. Step 8 just terminates the procedure.

9. Print the largest number among three numbers.

Solution Let the three numbers be represented by A , B , and C . There can be three ways of solving the problem. The three algorithms, with some differences, are given below.

1. START
2. PRINT "ENTER THREE NUMBERS"
3. INPUT A , B , C
4. IF $A \geq B$ AND $B \geq C$
 THEN PRINT A
5. IF $B \geq C$ AND $C \geq A$
 THEN PRINT B
- ELSE
 PRINT C
6. STOP

Explanation To find the largest among the three numbers A , B and C , A is compared with B to determine whether A is larger than or equal to B . At the same time it is also determined whether B is larger than or equal to C . If both these propositions are true then the number A is the largest otherwise A is not the largest. Step 4 applies this logic and prints A .

If A is not the largest number as found by the logic in step 4, then the logic stated in step 5 is applied. Here again, two propositions are compared. In one, B is compared with C and in the other C is compared with A . If both these propositions are true then B is printed as the largest otherwise C is printed as the largest.

Steps 1, 2, 3 and 6 needs no mention as it has been used in earlier examples.

Or

This algorithm uses a variable MAX to store the largest number.

1. START
2. PRINT "ENTER THREE NUMBERS"
3. INPUT A , B , C
4. $MAX \leftarrow A$
5. IF $B > MAX$ THEN $MAX \leftarrow B$
6. IF $C > MAX$ THEN $MAX \leftarrow C$
7. PRINT MAX
8. STOP

Explanation This algorithm differs from the previous one. After the numbers are stored in the variables A , B and C , the value of any one of these is assigned to a variable MAX . This is done in step 4. In step 5, the value assigned to MAX is compared with

that assigned to B and if the value in B is larger only then it's value is assigned to MAX otherwise it remains unchanged. In step 6, the proposition "IF $C > MAX$ " is true then the value in C is assigned to MAX. On the other hand, if the position is false then the value in MAX remains unchanged. So at the end of step 6, the value in MAX is the largest among the three numbers. Step 1 is the beginning step while step 8 is the terminating one for this algorithm.

Or

Here, the algorithm uses a **nested if** construct.

1. START
2. PRINT "ENTER THREE NUMBERS"
3. INPUT A, B, C
4. IF $A > B$ THEN
 - IF $A > C$ THEN
 - PRINT A
 - ELSE
 - PRINT C
 - ELSE IF $B > C$ THEN
 - PRINT B
 - ELSE
 - PRINT C
5. STOP

Explanation Here, the nested if construct is used. The construct "IF p1 THEN action1 ELSE action2" decides if the proposition "p1" is true then action1 is implemented otherwise if it is false action2 is implemented. Now, action1 and action2 may be either plain statements like PRINT X or INPUT X or another "IF p2 THEN action3 ELSE action4" construct, where p2 is a proposition. This means that a second "IF p1 THEN action1 ELSE action2" construct can be interposed within the first "IF p1 THEN action1 ELSE action2" construct. Such an implementation is known as nested if construct.

Step 4 implements the nested if construct. First the proposition " $A > B$ " is checked to find whether it is true or false. If true, the proposition " $A > C$ " is verified and if this is found to be true, the value in A is printed otherwise C is printed. But if the first proposition " $A > B$ " is found to be false then the next proposition that is checked is " $B > C$ ". At this point if this proposition is true then the value in B is printed whereas if it is false C is printed.

10. Take three sides of a triangle as input and check whether the triangle can be drawn or not. If possible, classify the triangle as equilateral, isosceles, or scalene.

Solution Let the length of three sides of the triangle be represented by A, B, and C. Two alternative algorithms for solving the problem are given, with explanations after each step, as follows:

1. START
 - Step 1 starts the procedure.
2. PRINT "ENTER LENGTH OF THREE SIDES OF A TRIANGLE"
 - Step 2 outputs a message asking for the entry of the lengths for each side of the triangle.
3. INPUT A, B, C
 - Step 3 reads the values for the lengths that has been entered and assigns them to A, B and C.
4. IF $A + B > C$ AND $B + C > A$ AND $A + C > B$ THEN
 - PRINT "TRIANGLE CAN BE DRAWN"
 - ELSE
 - PRINT "TRIANGLE CANNOT BE DRAWN": GO TO 6

It is well known that in a triangle, the summation of lengths of any two sides is always greater than the length of the third side. This is checked in step 4. So for a triangle all the propositions " $A + B > C$ ", " $B + C > A$ " and " $A + C > B$ " must be true. In such a case, with the lengths of the three sides, that has been entered, a triangle can be formed. Thus, the message "TRIANGLE CAN BE DRAWN" is printed and the next step 5 is executed. But if any one of the above three propositions is not true then the message "TRIANGLE CANNOT BE DRAWN" is printed and so no classification is required. Thus in such a case the algorithm is terminated in step 6.

5. IF $A = B$ AND $B = C$ THEN
 - PRINT "EQUILATERAL"
 - ELSE
 - IF $A \neq B$ AND $B \neq C$ AND $C \neq A$ THEN
 - PRINT "SCALENE"
 - ELSE
 - PRINT "ISOSCELES"

After it has been found in step 4 that a triangle can be drawn, this step is executed. To find whether the triangle is an "EQUILATERAL" triangle the propositions " $A = B$ " and " $B = C$ " are checked. If both of these are true, then the message "EQUILATERAL" is printed which means that the triangle is an equilateral triangle. On the other hand if any or both the propositions " $A = B$ " and " $B = C$ " are found to be untrue then the propositions " $A \neq B$ " and " $B \neq C$ " and " $C \neq A$ " are checked.

If none of the sides are equal to each other then all these propositions are found to be true and so the message "SCALENE" will be printed. But if these propositions "A != B" and "B != C" and "C != A" are false then the triangle is obviously an isosceles triangle and hence the message "ISOSCELES" is printed.

6. STOP

The procedure terminates here.

Or

This algorithm differs from the previous one and applies an alternate way to test whether a triangle can be drawn with the given sides and also identify its type.

1. START
2. PRINT "ENTER THE LENGTH OF 3 SIDES OF A TRIANGLE"
3. INPUT A, B, C
4. IF $A + B > C$ AND $B + C > A$ AND $C + A > B$ THEN
PRINT "TRIANGLE CAN BE DRAWN"
- ELSE
PRINT "TRIANGLE CANNOT BE DRAWN"
: GO TO 8
5. IF $A = B$ AND $B = C$ THEN
PRINT "EQUILATERAL TRIANGLE"
: GO TO 8
6. IF $A = B$ OR $B = C$ OR $C = A$ THEN
PRINT "ISOSCELES TRIANGLE"
: GO TO 8
7. PRINT "SCALENE TRIANGLE"
8. STOP

Having followed the explanations given with each of the earlier examples, the reader has already understood how the stepwise representation of any algorithm of any problem starts, constructs the logic statements and terminates.

In a similar way the following example exhibits the stepwise representation of algorithms for various problems.

11. In an academic institution, grades have to be printed for students who appeared in the final exam. The criteria for allocating the grades against the percentage of total marks obtained are as follows.

Marks	Grade	Marks	Grade
91–100	O	61–70	B
81–90	E	51–60	C
71–80	A	≤ 50	F

The percentage of total marks obtained by each student in the final exam is to be given as input to get a printout of the grade the student is awarded.

Solution The percentage of marks obtained by a student is represented by N . The algorithm for the given problem is as follows.

1. START
2. PRINT
"ENTER THE OBTAINED PERCENTAGE MARKS"
3. INPUT N
4. IF $N > 0$ AND $N \leq 50$ THEN
PRINT "F"
5. IF $N > 50$ AND $N \leq 60$ THEN
PRINT "C"
6. IF $N > 60$ AND $N \leq 70$ THEN
PRINT "B"
7. IF $N > 70$ AND $N \leq 80$ THEN
PRINT "A"
8. IF $N > 80$ AND $N \leq 90$ THEN
PRINT "E"
9. IF $N > 90$ AND $N \leq 100$ THEN
PRINT "O"
10. STOP

12. Construct an algorithm for incrementing the value of a variable that starts with an initial value of 1 and stops when the value becomes 5.

Solution This problem illustrates the use of iteration or loop construct. Let the variable be represented by C . The algorithm for the said problem is given as follows.

1. START
2. $C \leftarrow 1$

```

3. WHILE  $C \leq 5$ 
4. BEGIN
5. PRINT C
6.  $C \leftarrow C + 1$ 
7. END

```

While loop construct for looping till C is greater than 5

8. STOP

13. Write an algorithm for the addition of N given numbers.

Solution Let the sum of N given numbers be represented by S . Each time a number is given as input, it is assigned to the variable A . The algorithm using the loop construct 'if ... then goto ...' is used as follows:

1. START
2. PRINT "HOW MANY NUMBERS?"
3. INPUT N
4. $S \leftarrow 0$
5. $C \leftarrow 1$
6. PRINT "ENTER NUMBER"
7. INPUT A
8. $S \leftarrow S + A$
9. $C \leftarrow C + 1$

```

10. IF C <= N THEN GOTO 6
11. PRINT S
12. STOP

```

14. Develop the algorithm for finding the sum of the series $1 + 2 + 3 + 4 + \dots$ up to N terms.

Solution Let the sum of the series be represented by S and the number of terms by N . The algorithm for computing the sum is given as follows.

```

1. START
2. PRINT "HOW MANY TERMS?"
3. INPUT N
4. S ← 0
5. C ← 1
6. S ← S + C
7. C ← C + 1
8. IF C <= N THEN GOTO 6
9. PRINT S
10. STOP

```

15. Write an algorithm for determining the sum of the series $2 + 4 + 8 + \dots$ up to N .

Solution Let the sum of the series be represented by S and the number of terms in the series by N . The algorithm for this problem is given as follows.

```

1. START
2. PRINT "ENTER THE VALUE OF N"
3. INPUT N
4. S ← 0
5. C ← 2
6. S ← S + C
7. C ← C * 2
8. IF C <= N THEN GOTO STEP 6
9. PRINT S
10. STOP

```

16. Write an algorithm to find out whether a given number is a prime number or not.

Solution The algorithm for checking whether a given number is a prime number or not is as follows.

```

1. START
2. PRINT "ENTER THE NUMBER"
3. INPUT N
4. IF N = 2 THEN
    PRINT "CO-PRIME" GOTO STEP 12
5. D ← 2
6. Q ← N/D (Integer division)
7. R ← N - Q*D

```

```

8. IF R = 0 THEN GOTO STEP 11
9. D ← D + 1
10. IF D <= N/2 THEN GOTO STEP 6
11. IF R = 0 THEN
    PRINT "NOT PRIME"
ELSE
    PRINT "PRIME"
12. STOP

```

17. Write an algorithm for calculating the factorial of a given number N .

Solution The algorithm for finding the factorial of number N is as follows.

```

1. START
2. PRINT "ENTER THE NUMBER"
3. INPUT N
4. F ← 1
5. C ← 1
6. WHILE C <= N
7. BEGIN
8. F ← F * C
9. C ← C + 1
10. END
11. PRINT F
12. STOP

```

While loop construct for looping till C is greater than N

18. Write an algorithm to print the Fibonacci series up to N terms.

Solution The Fibonacci series consisting of the following terms $1, 1, 2, 3, 5, 8, 13, \dots$ is generated using the following algorithm.

```

1. START
2. PRINT "ENTER THE NUMBER OF TERMS"
3. INPUT N
4. C ← 1
5. T ← 1
6. T1 ← 0
7. T2 ← 1
8. PRINT T
9. T ← T1 + T2
10. C ← C + 1
11. T1 ← T2
12. T2 ← T
13. IF C <= N THEN GOTO 8
14. STOP

```

19. Write an algorithm to find the sum of the series $1 + x + x^2 + x^3 + x^4 + \dots$ up to N terms.

Solution

```

1. START
2. PRINT "HOW MANY TERMS"
3. INPUT N

```

4. PRINT "ENTER VALUE OF X"
5. INPUT X
6. $T \leftarrow 1$
7. $C \leftarrow 1$
8. $S \leftarrow 0$
9. $S \leftarrow S + T$
10. $C \leftarrow C + 1$
11. $T \leftarrow T * X$
12. IF $C \leq N$ THEN GOTO 9
13. PRINT S
14. STOP

20. Write the algorithm for computing the sum of digits in a number.

Solution

1. START
2. PRINT "ENTER THE NUMBER"
3. INPUT N
4. $S \leftarrow 0$
5. $Q \leftarrow N/10$ (Integer division)
6. $R \leftarrow N - Q * 10$
7. $S \leftarrow S + R$
8. $N \leftarrow Q$
9. IF $N > 0$ THEN GOTO 5
10. PRINT S
11. STOP

21. Write an algorithm to find the largest number among a list of numbers.

Solution The largest number can be found using the following algorithm.

1. START
2. PRINT "ENTER,
TOTAL COUNT OF NUMBERS IN LIST"
3. INPUT N
4. $C \leftarrow 0$
5. PRINT "ENTER THE NUMBER"
6. INPUT A
7. $C \leftarrow C + 1$
8. $MAX \leftarrow A$
9. PRINT "ENTER THE NUMBER"
10. INPUT B
11. $C \leftarrow C + 1$
12. IF $B > MAX$ THEN
 $MAX \leftarrow B$
13. IF $C \leq N$ THEN GOTO STEP 9
14. PRINT MAX
15. STOP

22. Write an algorithm to check whether a given number is an Armstrong number or not. An Armstrong number is one in which the sum of the cube of each of the digits equals that number.

Solution If a number 153 is considered, the required sum is $(1^3 + 5^3 + 3^3)$, i.e., 153. This shows that the number is an Armstrong number. The algorithm to check whether 153 is an Armstrong number or not, is given as follows.

1. START
2. PRINT "ENTER THE NUMBER"
3. INPUT N
4. $M \leftarrow N$
5. $S \leftarrow 0$
6. $Q \leftarrow N/10$ (Integer division)
7. $R \leftarrow N - Q * 10$
8. $S \leftarrow S + R * R * R$
9. $N \leftarrow Q$
10. IF $N > 0$ THEN GOTO STEP 6
11. IF $S = M$ THEN
PRINT "THE NUMBER IS ARMSTRONG"
- ELSE PRINT "THE NUMBER IS NOT ARMSTRONG"
12. STOP

23. Write an algorithm for computing the sum of the series $1 + x + x^2/2! + x^3/3! + x^4/4! + \dots$ up to N terms.

Solution

1. START
2. PRINT "ENTER NUMBER OF TERMS"
3. INPUT N
4. PRINT "ENTER A NUMBER"
5. INPUT X
6. $T \leftarrow 1$
7. $S \leftarrow 0$
8. $C \leftarrow 1$
9. $S \leftarrow S + T$
10. $T \leftarrow T * X/C$
11. $C \leftarrow C + 1$
12. IF $C \leq N$ THEN GO TO STEP 9
13. PRINT S
14. STOP

Pseudo-code

Like step-form, Pseudo-code is a written statement of an algorithm using a restricted and well-defined vocabulary. It is similar to a 3GL, and for many programmers and program designers it is the preferred way to state algorithms and program specifications.

Although there is no standard for pseudo-code, it is generally quite easy to read and use. For instance, a sample pseudo-code is written as follows:

```
dowhile kettle_empty
  Add_Water_To_Kettle
end dowhile
```

As can be seen, it is a precise statement of a while loop.

Flowcharts

A flowchart provides appropriate steps to be followed in order to arrive at the solution to a problem. It is a program design tool which is used before writing the actual program. Flowcharts are generally developed in the early stages of formulating computer solutions.

A flowchart comprises a set of various standard shaped boxes that are interconnected by flow lines. Flow lines have arrows to indicate the direction of the flow of control between the boxes. The activity to be performed is written within the boxes in English. In addition, there are connector symbols that are used to indicate that the flow of control continues elsewhere, for example, the next page.

Flowcharts facilitate communication between programmers and business persons. These flowcharts play a vital role in the programming of a problem and are quite helpful in understanding the logic of complicated and lengthy problems. Once the flowchart is drawn, it becomes easy to write the program in any high-level language. Often flowcharts are helpful in explaining the program to others. Hence, a flowchart is a must for better documentation of a complex program.

Standards for flowcharts The following standards should be adhered to while drawing flow charts.

- Flowcharts must be drawn on white, unlined 8 1/2 × 11 paper, on one side only.
- Flowcharts start on the top of the page and flow down and to the right.
- Only standard flowcharting symbols should be used.
- A template to draw the final version of flowchart should be used.
- The contents of each symbol should be printed legibly.
- English should be used in flowcharts, not programming language.

- The flowchart for each subroutine, if any, must appear on a separate page. Each subroutine begins with a terminal symbol with the subroutine name and a terminal symbol labeled return at the end.
- Draw arrows between symbols with a straight edge and use arrowheads to indicate the direction of the logic flow.

Guidelines for drawing a flowchart Flowcharts are usually drawn using standard symbols; however, some special symbols can also be developed when required. Some standard symbols frequently required for flowcharting many computer programs are shown in Fig.1.12.

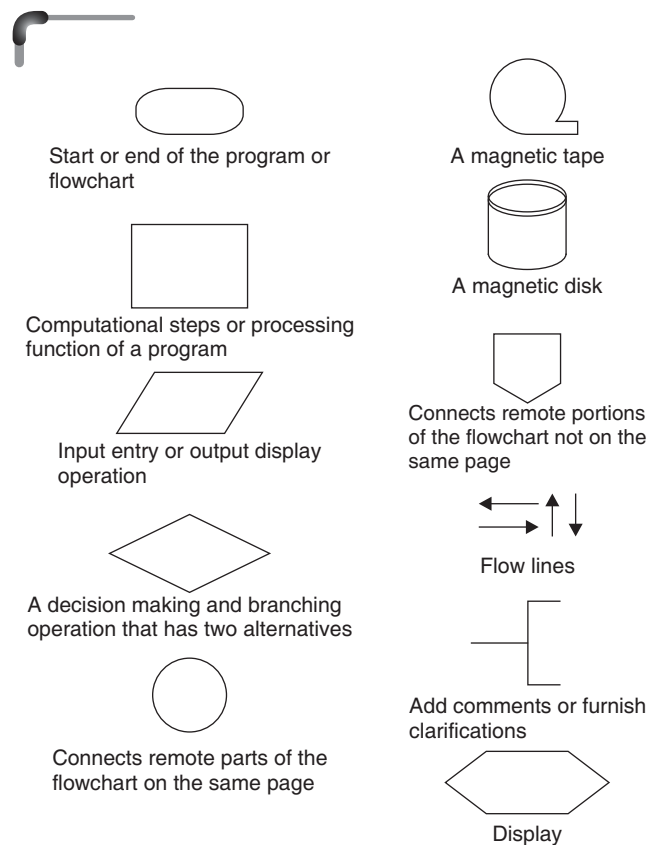
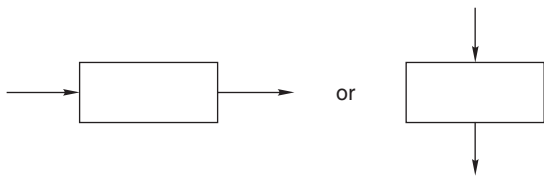


Figure 1.12 Flowchart symbols

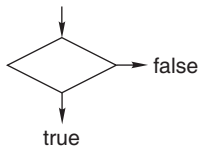
The following are some guidelines in flowcharting.

- In drawing a proper flowchart, all necessary requirements should be listed out in a logical order.
- There should be a logical **start** and **stop** to the flowchart.

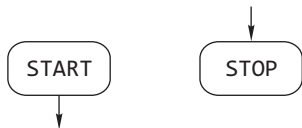
- The flowchart should be clear, neat, and easy to follow. There should be no ambiguity in understanding the flowchart.
- The usual direction of the flow of a procedure or system is from left to right or top to bottom.
- Only one flow line should emerge from a process symbol.



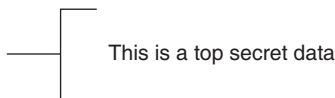
- Only one flow line should enter a decision symbol, but two or three flow lines, one for each possible answer, can leave the decision symbol.



- Only one flow line is used in conjunction with a terminal symbol.

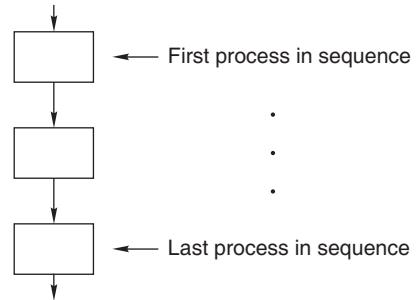


- The writing within standard symbols should be brief. If necessary, the annotation symbol can be used to describe data or computational steps more clearly.

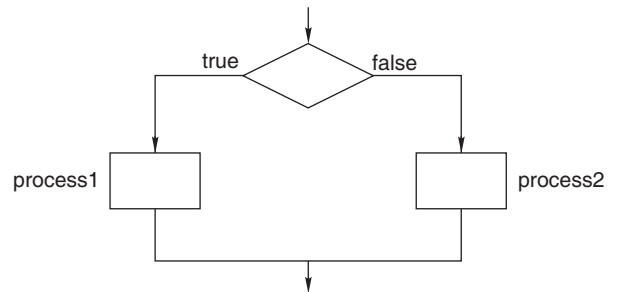


- If the flowchart becomes complex, connector symbols should be used to reduce the number of flow lines. The intersection of flow lines should be avoided to make the flowchart a more effective and better way of communication.

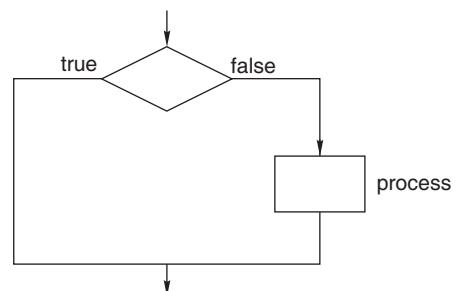
- The validity of the flowchart should be tested by passing simple test data through it.
- A *sequence* of steps or processes that are executed in a particular order is shown using process symbols connected with flow lines. One flow line enters the first process while one flow line emerges from the last process in the sequence.



- *Selection* of a process or step is depicted by the decision making and process symbols. Only one input indicated by one incoming flow line and one output flowing out of this structure exists. The decision symbol and the process symbols are connected by flow lines.



- *Iteration* or *looping* is depicted by a combination of process and decision symbols placed in proper order. Here flow lines are used to connect the symbols and depict input and output to this structure.



Advantages of using flowcharts

- *Communication*: Flowcharts are a better way of communicating the logic of a system to all concerned.
- *Effective analysis*: With the help of flowcharts, problems can be analyzed more effectively.
- *Proper documentation*: Program flowcharts serve as a good program documentation needed for various purposes.
- *Efficient coding*: Flowcharts act as a guide or blueprint during the systems analysis and program development phase.
- *Proper debugging*: Flowcharts help in the debugging process.
- *Efficient program maintenance*: The maintenance of an operating program becomes easy with the help of a flowchart.

Limitations of using flowcharts

- *Complex logic*: Sometimes, the program logic is quite complicated. In such a case, a flowchart becomes complex and clumsy.
- *Alterations and modifications*: If alterations are required, the flowchart may need to be redrawn completely.
- *Reproduction*: Since the flowchart symbols cannot be typed in, the reproduction of a flowchart becomes a problem.
- *Loss of objective*: The essentials of what has to be done can easily be lost in the technical details of how it is to be done.

Points to Note

1. A flowchart comprises a set of standard shaped boxes that are interconnected by flow lines to represent an algorithm.
2. There should be a logical start and stop to the flowchart.
3. The usual direction of the flow of a procedure or system is from left to right or top to bottom.
4. The intersection of flow lines should be avoided.
5. Flowcharts facilitate communication between programmers and users.

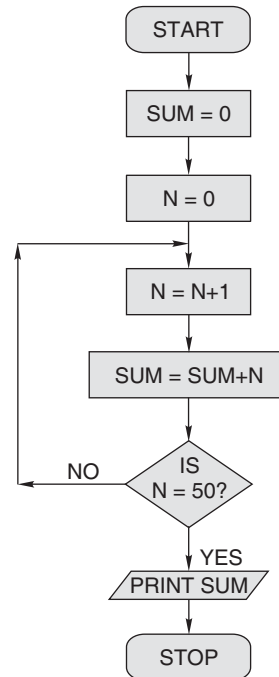
Flowcharting examples A few examples on flowcharting are presented for a proper understanding of the technique.

This will help the student in the program development process at a later stage.

Examples

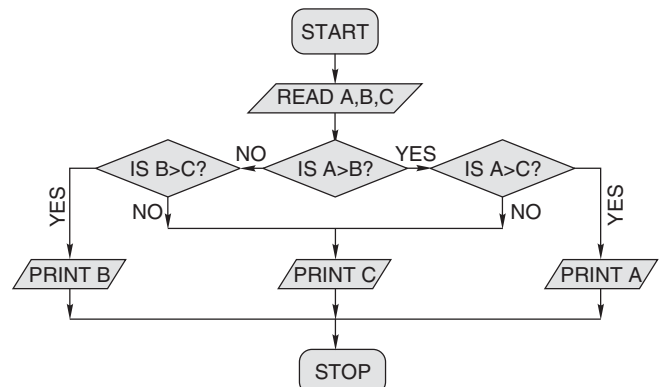
24. Draw a flowchart to find the sum of the first 50 natural numbers.

Solution



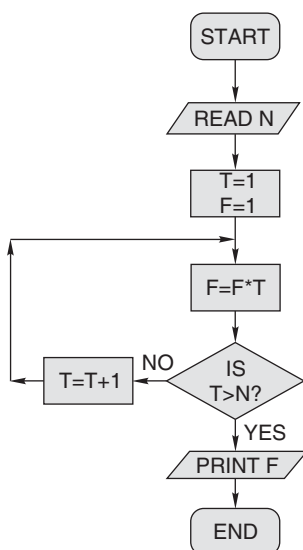
25. Draw a flowchart to find the largest of three numbers A, B, and C.

Solution



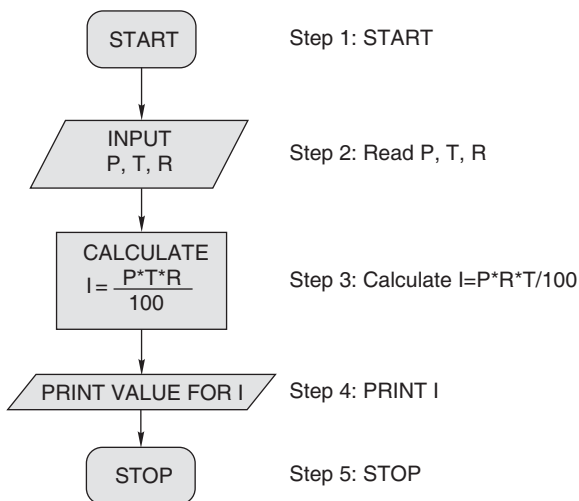
26. Draw a flowchart for computing factorial N ($N!$) where $N! = 1 \times 2 \times 3 \times \dots \times N$.

Solution



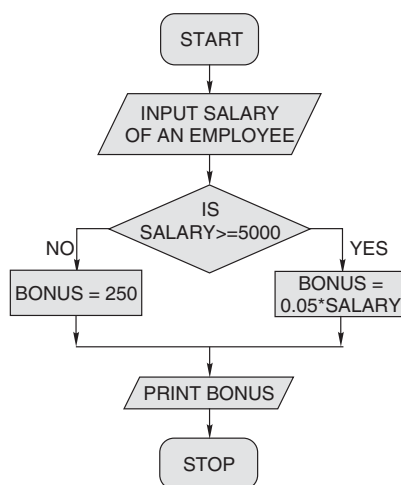
27. Draw a flowchart for calculating the simple interest using the formula $SI = (P * T * R)/100$, where P denotes the principal amount, T time, and R rate of interest. Also, show the algorithm in step-form.

Solution



28. The XYZ Construction Company plans to give a 5% year-end bonus to each of its employees earning Rs 5,000 or more per year, and a fixed bonus of Rs 250 to all other employees. Draw a flowchart and write the step-form algorithm for printing the bonus of any employee.

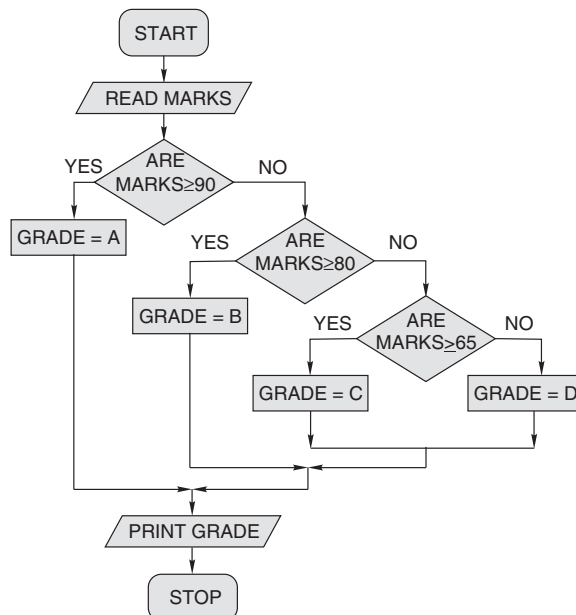
Solution



Step 1: START
 Step 2: Read salary of an employee
 Step 3: IF salary is greater than or equal to 5,000 THEN Step 4 ELSE Step 5
 Step 4: Calculate Bonus = 0.05 * Salary
 Step 5: Calculate Bonus = 250
 Step 6: Print Bonus
 Step 7: STOP

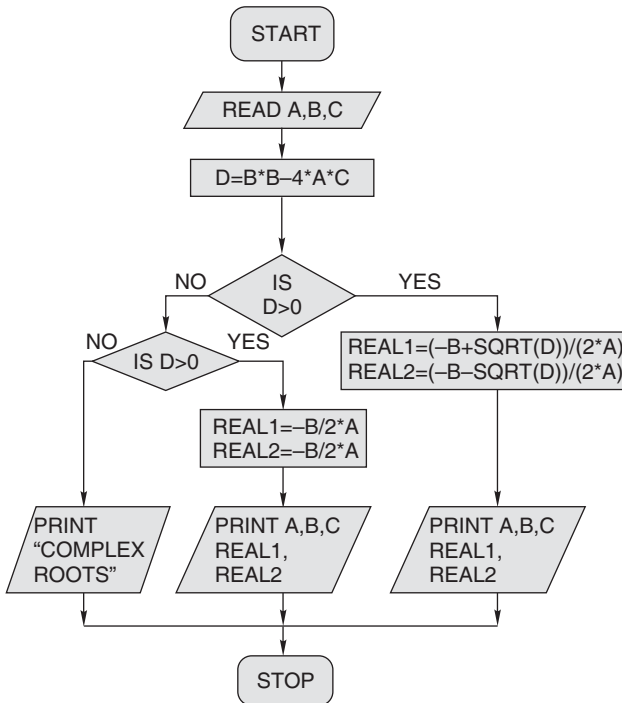
29. Prepare a flowchart to read the marks of a student and classify them into different grades. If the marks secured are greater than or equal to 90, the student is awarded Grade A; if they are greater than or equal to 80 but less than 90, Grade B is awarded; if they are greater than or equal to 65 but less than 80, Grade C is awarded; otherwise Grade D is awarded.

Solution



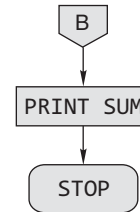
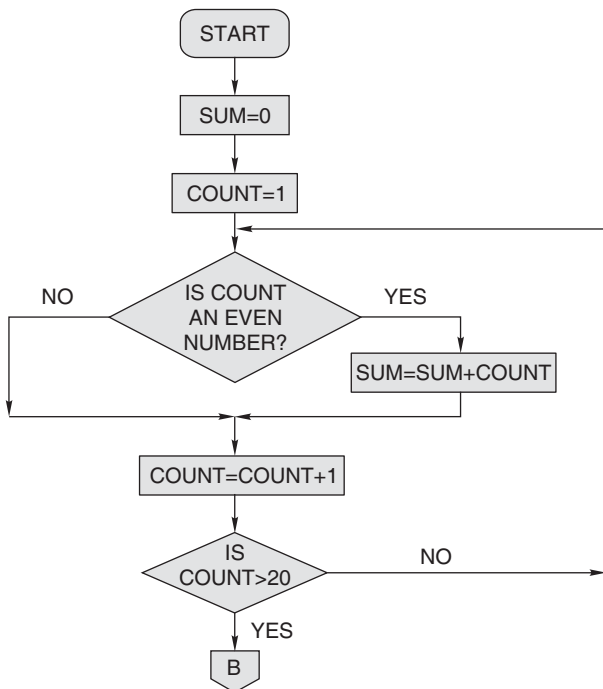
30. Draw a flowchart to find the roots of a quadratic equation.

Solution



31. Draw a flowchart for printing the sum of even terms contained within the numbers 0 to 20.

Solution



1.9.6 Strategy for Designing Algorithms

Now that the meaning of algorithm and data has been understood, strategies can be devised for designing algorithms. The following is a useful strategy.

Investigation step

1. *Identify the outputs needed.*

This includes the form in which the outputs have to be presented. At the same time, it has to be determined at what intervals and with what precision the output data needs to be given to the user.

2. *Identify the input variables available.*

This activity considers the specific inputs available for this program, the form in which the input variables would be available, the availability of inputs at different intervals, the ways in which the input would be fed to the transforming process.

3. *Identify the major decisions and conditions.*

This activity looks into the conditions imposed by the need identified and the limitations of the environment in which the algorithm has to be implemented.

4. *Identify the processes required to transform inputs into required outputs.*

This activity identifies the various types of procedures needed to manipulate the inputs, within the bounding conditions and the limitations mentioned in step 3, to produce the needed outputs.

5. *Identify the environment available.*

This activity determines the kind of users and the type of computing machines and software available for implementing the solution through the processes considered in steps.

Top-down development step

1. *Devise the overall problem solution by identifying the major components of the system.*

The goal is to divide the problem solution into manageable small pieces that can be solved separately.

2. *Verify the feasibility of breaking up the overall problem solution.*

The basic idea here is to check that though each small piece of solution procedure are independent, they are not entirely independent of each other, as they together form the whole solution to the problem. In fact, the different pieces of solution procedures have to cooperate and communicate in order to solve the larger problem.

Stepwise refinement

1. *Work out each and every detail for each small piece of manageable solution procedure.*

Every input and output dealt with and the transformation algorithms implemented in each small piece of solution procedure, which is also known as process, is detailed. Even the interfacing details between each small procedure are worked out.

2. *Decompose any solution procedure into further smaller pieces and iterate until the desired level of detail is achieved.*

Every small piece of solution procedure detailed in step 1 is checked once again. If necessary any of these may be further broken up into still smaller pieces of solution procedure till it can no more be divided into meaningful procedure.

3. *Group processes together which have some commonality.*
Some small processes may have to interface with a common upper level process. Such processes may be grouped together if required.

4. *Group variables together which have some appropriate commonality.*

Certain variables of same type may be dealt as elements of a group.

5. *Test each small procedure for its detail and correctness and its interfacing with the other small procedures.*

Walk through each of the small procedures to determine whether it satisfies the primary requirements and would deliver the appropriate outputs. Also, suitable tests have to be carried out to verify the interfacing

between various procedures. Hence, the top-down approach starts with a big and hazy goal. It breaks the big goal into smaller components. These components are themselves broken down into smaller parts. This strategy continues until the designer reaches the stage where he or she has concrete steps that can actually be carried out.

It has to be noted that the top-down approach does not actually take into account any existing equipment, people, or processes. It begins with a “clean slate” and obtains the optimal solution. The top-down approach is most appropriate for large and complex projects where there is no existing equipment to worry about. However, it may be costly because, sometimes, the existing equipments may not fit into the new plan and it has to be replaced. However, if the existing equipments can be made to fit into the new plan with very less effort, it would be beneficial to use it and save cost.

Points to Note

1. Investigation phase determines the requirements for the problem solution.
2. The top-down development phase plans out the way the solution has to be done by breaking it into smaller modules and establishing a logical connection among them.
3. The step-wise refinement further decomposes the modules, defines the procedure in it and verifies the correctness of it.

1.9.7 Tracing an Algorithm to Depict logic

An algorithm is a collection of some procedural steps that have some precedence relation between them. Certain procedures may have to be performed before some others are performed. Decision procedures may also be involved to choose whether some procedures arranged one after other are to be executed in the given order or skipped or implemented repetitively on fulfillment of conditions arising out of some preceding manipulations. Hence, an algorithm is a collection of procedures that results in providing a solution to a problem. *Tracing* an algorithm primarily involves tracking the outcome of every procedure in the order they are placed. *Tracking* in turn means verifying every procedure one by one to determine

and confirm the corresponding result that is to be obtained. This in turn can be traced to offer an overall output from the implementation of the algorithm as a whole. Consider Example 26 given in this chapter for the purpose of tracing the algorithm to correctly depict the logic of the solution. Here at the start, the “mark obtained by a student in a subject” is accepted as input to the algorithm. This procedure is determined to be essential and alright. In the next step, the marks entered is compared with 90. As given, if the mark is greater than 90, then the mark obtained is categorized as Grade A and printed, otherwise it is further compared. Well, this part of the algorithm matches with the requirement and therefore this part of the logic is correct.

For the case of further comparison, the mark is again compared with 80 and if it is greater, then Grade B is printed. Otherwise, if the mark is less than 80, then further comparison is carried out. This part of the logic satisfies the requirement of the problem. In the next step of comparison, the mark is compared with 65. If the mark is lesser than 65, Grade C is printed, otherwise Grade D is printed. Here also, the flowchart depicts that the correct logic has been implemented.

The above method shows how the logic of an algorithm, planned and represented by a tool like the flowchart, can be verified for its correctness. This technique, also referred to as *deskcheck* or *dry run*, can also be used for algorithms represented by tools other than the flowchart.

1.9.8 Specification for Converting Algorithms into Programs

By now, the method of formulating an algorithm has been understood. Once the algorithm, for solution of a problem, is formed and represented using any of the tools like step-form or flowchart or pseudo code, etc., it has to be transformed into some programming language code. This means that a program, formed by a sequence of program instructions belonging to a programming language, has to be written to represent the algorithm that provides a solution to a problem.

Hence, the general procedure to convert an algorithm into a program is given as follows:

Code the algorithm into a program—Understand the syntax and control structures used in the language that has been selected and write the equivalent program instructions based upon the algorithm that was created.

Each statement in an algorithm may require one or more lines of programming code.

Desk-check the program—Check the program code by employing the desk-check method and make sure that the sample data selected produces the expected output.

Evaluate and modify, if necessary, the program—Based on the outcome of desk-checking the program, make program code changes, if necessary, or make changes to the original algorithm, if need be.

Do not reinvent the wheel—If the design code already exists, modify it, do not remake it.

Points to Note

1. An algorithm can be traced by verifying every procedure one by one to determine and confirm the corresponding result that is to be obtained.
2. The general procedure to convert an algorithm into a program is to code the algorithm using a suitable programming language, check the program code by employing the desk-check method and finally evaluate and modify the program, if needed.

Because the reader has not yet been introduced to the basics of the C language, the reader has to accept the use of certain instructions like `#include <stdio.h>`, `int main()`, `printf()`, `scanf()`, and `return` without much explanation at this stage in the example program being demonstrated below.

However, on a very preliminary level, the general form of a C program and the use of some of the necessary C language instructions are explained briefly as follows:

1. All C programs start with:

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

2. In C, all variables must be declared before using them. So the line next to the two instruction lines, given in step 1, should be any variable declarations that is needed.

For example, if a variable called “a” is supposed to store an integer, then it is declared as follows:

```
int a;
```

3. Here, `scanf()` is used for inputting data to the C program and `printf()` is used to output data on the monitor screen.

4. The C program has to be terminated with a statement given below:

```
return 0;
}
```

Here is an example showing how to convert some pseudocode statements into C language statements:

Pseudocode

```
LOOP {
EXIT LOOP
IF (conditions) {
ELSE IF (conditions) {
ELSE {
INPUT a
OUTPUT "Value of a:" a
+ - * / %
=
<-
!=
AND
OR
NOT
```

C language Code

```
while(1) {
break;
if (conditions) {
else if (conditions) {
else
scanf("%d",&a);
printf("Value of a: %d",a);
(same)
==
```

```
=
!=
&&
||
!
```

To demonstrate the procedure of conversion from an algorithm to a program in C, an example is given below.

Problem statement Write the algorithm and the corresponding program in C for adding two integer numbers and printing the result.

Solution

Algorithm

1. START
2. DECLARE A AND B AS INTEGER VARIABLES
3. PRINT " ENTER TWO NUMBERS "
4. INPUT A, B
5. R = A + B
6. PRINT " RESULT = "
7. PRINT R
8. STOP.

Program in C


```
int main( )
{
int A, B;
printf("\n ENTER TWO NUMBERS:");
scanf("%d%d",&A,&B);
R = A + B;
printf("\n RESULT = ");
printf("%d",R);
return 0;
}
```




SUMMARY

A program is a sequence of instructions and the process of writing a program is called programming. Programs are broadly categorized as system programs and application programs. Different programming languages have evolved. High-level languages are easy to use while low-level languages are complex. Therefore, writing programs in low-level languages is difficult and time consuming.

Compilers and interpreters are basically language translators that convert program instructions to machine code. A linker attaches utilities

to the translated source code. A loader is responsible for physically placing this code in the main memory.

An algorithm is a statement about how a problem will be solved and almost every algorithm exhibits the same features. There are many ways of stating algorithms; three of them have been mentioned here. These are step-form, pseudo code, and flowchart method. Of these flowchart is a pictorial way of representing the algorithm. Here, the START and STOP are represented by an ellipse-like figure, , decision

construct by the rhombus-like figure, , the processes by rectangles,  and input/output by parallelograms, . Lines and arrows connect these blocks. Every useful algorithm uses data, which might vary during the course of the algorithm. To design algorithms, it is a good idea to develop and use a design strategy.

Generally the design strategy consists of three stages. The first stage is investigation activity followed by the top-down development approach stage and eventually a stepwise refinement process. Once the design strategy is decided the algorithm designed is traced to determine whether it represents the logic. Eventually, the designed and checked, algorithm is transformed into a program.

KEY-TERMS

Algorithm Specifies a procedure for solving a problem in a finite number of steps.

Application software A collection of programs that enables the computer to solve a specific data processing task.

Assembler A translator that takes input in the form of the assembly language and produces machine language code as its output.

Assembly language A low-level programming language.

Compiler A language translator that takes the high-level language program as input and produces the executable machine language code.

Correctness Means how easily its logic can be argued to meet the algorithm's primary goal.

Data A symbolic representation of value.

Debug To search and remove errors in a program.

High-level programming language A language similar to human languages that makes it easy for a programmer to write programs and identify and correct errors in them.

Interpreter A language translator that translates and executes a program line by line.

Investigation step A step to determine the input, output and processing requirements of a problem.

Linker A program that resolves references between programs.

Loader A program that physically places the machine instructions and data in main memory.

Low-level programming language Closer to the native language of the computer, which is 1's and 0's.

Machine language Language that provides instructions in the form of binary numbers consisting of 1's and 0's to which the computer responds directly

Operating system System software that manages the computer's resources effectively.

Portability language Programming language that is not machine dependent and can be used in any computer.

Program A set of logically related instructions arranged in a sequence that directs the computer in solving a problem.

Programming languages A language composed of a set of instructions understandable by the programmer.

Programming The process of writing a program.

System software A collection of programs that interfaces with the computer hardware.

Termination Closure of a procedure.

Top-down analysis Breaking up a problem solution into smaller modules and defining their interconnections to provide the total solution to a problem.

Variable A container for a value that may or may not vary during the execution of the program.

FREQUENTLY ASKED QUESTIONS

1. What is a programming language?

A programming language is an artificial formalism in which algorithms can be expressed. More formally, a *computer program* is a sequence of instructions that is used to operate a computer to produce a specific result.

A programming language is the communication bridge between a programmer and computer. A programming language allows a programmer to create sets of executable instructions called programs that the computer can understand. This communication bridge is needed because computers understand only machine language, which is an instruction language in which data are represented by binary digits.

2. What is a token?

A token is any word or symbol that has meaning in the language, such as a keyword (reserved word) such as *if* or *while*. The tokens are *parsed* or grouped according to the rules of the language.

3. What is syntax?

Syntax is the 'grammar' of the programming language. It specifies the formal rules governing the way the vocabulary elements of the language can be combined to form instructions. The syntax of a programming language defines exactly what combinations of letters, numbers, and symbols can be used in a programming language. During compilation, all syntax rules are checked. If a program is not syntactically correct, the compiler will issue error messages and will not produce object code.

4. What is a variable?

A *variable* is a name given to the area of computer memory that holds the relevant data. Each variable has a data type, which might be number, character, string, a collection of data elements (such as an array), a data record, or some special type defined by the programmer.

5. What are the difficulties faced in procedural programming?

The main drawback of procedural programming is that it breaks down when problems become very large especially when it is highly complex, making it somewhat more difficult for a team of people to work with it. There are limits to the amount of detail and largeness one can cope with. Non-procedural programming like object-oriented programming can help the programmer compartmentalize and manage that detail. Various forms of non-procedural programming are vastly more effective for many large real-world problems.

6. What is Spaghetti code?

Non-modular code is normally referred to as spaghetti code. It is named so because it produces a disorganized computer program using many GOTO statements.

7. What is structured programming?

Structured programming is a style of programming designed to make programs more comprehensible and programming errors less frequent. This technique of programming enforces a logical structure on the program being written to make it more efficient and easier to understand and modify. It usually includes the following characteristics:

Block structure The statements in the program must be organized into functional groups. It emphasizes clear logic.

Avoidance of jumps A lot of GOTO statements makes the programs more error-prone. Structured programming uses less of these statements. Therefore it is also known as 'GOTO less programming'.

Modularity It is a common idea that structuring the program makes it easier for us to understand and therefore easier for teams of developers to work simultaneously on the same program.

8. What are the advantages and disadvantages of structured programming?

Structured programming provides options to develop well-organized codes which can be easily modified and documented.

Modularity is closely associated with structured programming. The main idea is to structure the program into functional groups. As a result, it becomes easier for us to understand and therefore easier for teams of developers to work simultaneously on the same program.

Another advantage of structured programming is that it *reduces complexity*. Modularity allows the programmer to tackle problems in a logical fashion. This improves the programming process through better organization of programs and better programming notations to facilitate correct and clear description of data and control structure.

Structured programming also *saves time* as without modularity, the code that is used multiple times needs to be written every time it is used. On the other hand, modular programs need one to call a subroutine (or function) with that code to get the same result in a structured program.

Structured programming *encourages stepwise refinement*, a program design process described by Niklaus Wirth. This is a top-down approach in which the stages of processing are first described in high-

level terms, and then gradually worked out in their details, much like the writing of an outline for a book.

The disadvantages of structured programming include the following:

Firstly, error control may be harder to manage. Managing modifications may also be difficult.

Secondly, debugging efforts can be hindered because the problem code will look right and even perform correctly in one part of the program but not in another.

9. What is pseudocode?

Pseudocode is an informal description of a sequence of steps for solving a problem. It is an outline of a computer program, written in a mixture of a programming language and English. Writing pseudocodes is one of the best ways to plan a computer program.

The advantage of having pseudocodes is that it allows the programmer to concentrate on how the program works while ignoring the details of the language. By reducing the number of things the programmer must think about at once, this technique effectively amplifies the programmer's intelligence.

10. What is top-down programming?

Top-down programming is a technique of programming that first defines the overall outlines of the program and then fills in the details.

This approach is usually the best way to write complicated programs. Detailed decisions are postponed until the requirements of the large program are known; this is better than making the detailed decisions early and then forcing the major program strategy to conform to them. Each part of the program (called a *module*) can be written and tested independently.

11. What is an error? Describe different types of error that may occur in a program.

An error that occurs during the compilation stage is called a *compiler error*. A compiler error occurs when a given program does not follow the grammatical rules of a C program.

An error that occurs during the linking stage is called a *linker error*. A linker error typically occurs when the linker cannot locate the file to be linked.

Finally, an error that occurs during the execution of a program is called a *runtime error*. These are the most troublesome errors to correct.

Logic errors are errors in a program that executes without performing the intended action. In this case, the program compiles and executes without complaints, but it produces incorrect results. It occurs when the logic of the program as written is different from what was actually intended. A compiler cannot find such errors, and it must be flushed out when the program runs, by testing it and carefully looking at its output. The programmer is responsible for inspecting and testing the program to guard against logic errors.

12. What is a debugger?

A debugger is a programming tool that is used to debug a program, i.e., to correct the logical errors. Using a debugger, one can control a program while it is running. The execution of the program can be stopped at some point and the values in the different variables can be checked and these values can be amended if desired. In this way, the logical errors can be traced in the program and it can be seen whether the program is producing correct results. This tool is very powerful and complex.

13. What is the function of a loader?

After an executable program is linked and saved on the disk, it is ready for execution. A program called *loader* is needed to load the program into memory and then instruct the processor to execute the program from the first instruction (the starting point of every C program is from the main function). This processor is known as a loader. Linker and loaders are the parts of development environment. In fact, these are the parts of system software.

14. What do you mean by high-level and low-level programming languages? Differentiate between them.

Both assembly language and machine language are considered as *low-level languages*. The instructions in these languages have to take

into account the physical characteristics of the machine. Maybe these features are completely irrelevant to the algorithm, but they have to be considered while writing programs or developing algorithms.

High-level programming languages, on the other hand, are those which support the use of constructs that use appropriate abstraction mechanisms to ensure that they are independent of the physical characteristics of the computer. The term 'high-level' refers to the fact that the programming statements are expressed in a form approaching natural language, far removed from the machine language that is ultimately executed.

The difference between high level language and low level language is summarized in the following table.

High-level Language	Low-level Language
One instruction = many machine code instructions	One instruction = one machine code instruction
Portable, task-oriented	Machine specific, machine-oriented
More English-like	Less easy to write and debug

EXERCISE

- What do you mean by a program?
- Distinguish between system software and application software.
- State the advantages and disadvantages of machine language and assembly language.
- Compare and contrast assembly language and high-level language.
- Differentiate between 3GL and 4GL.
- What is a translator?
- What are the differences between a compiler and an interpreter?
- Briefly explain the compilation and execution of a program written in high-level language.
- Briefly explain linker and loader? Is there any difference between them?
- Explain linking loader and linkage editor?
- Classify the programming languages.
- What is a functional language?
- What is object-oriented language? Name five object-oriented programming languages. State the most common features of object-oriented programming.
- What do you mean by structured programming? State the properties of structured programming.
- What is top-down analysis? Describe the steps involved in top-down analysis.
- What is a structured code?
- What is an algorithm?
- Write down an algorithm that describes making a telephone call. Can it be done without using control statements?
- Write algorithms to do the following:
 - Check whether a year given by the user is a leap year or not.
 - Given an integer number in seconds as input, print the equivalent time in hours, minutes, and seconds as output. The recommended output format is something like:
7,322 seconds is equivalent to 2 hours 2 minutes 2 seconds.
 - Print the numbers that do not appear in the Fibonacci series. The number of terms to be printed should be given by the user.
 - Convert the binary equivalent of an integer number.
 - Find the prime factors of a number given by the user.
 - Check whether a number given by the user is a Krishnamurty number or not. A Krishnamurty number is one for which the sum of the factorials of its digits equals the number. For example, 145 is a Krishnamurty number.
 - Print the second largest number of a list of numbers given by the user.

- (h) Print the sum of the following series:
- $1/x^2! + x^4/4!$ up to n terms where n is given by the user
 - $1/2 + 1/3$ up to n terms where n is given by the user
 - $1 + 1/2! + 1/3! +$ up to n terms where n is given by the user

20. By considering the algorithmic language that has been taught, answer the following:

- (a) Show clearly the steps of evaluating the following expressions:
- $x + y + 12 * 3/6 + k^x$ where $x = 2, y = 6, k = 5$
 - $a \text{ AND } b \text{ OR } (m < n)$ where $a = \text{true}, b = \text{false}, m = 7, n = 9$
- (b) State whether each of the following is correct or wrong. Correct the error(s) where applicable.
- The expression $(35 = 035)$ is true.
 - $x1 \ x2 * 4$ value
 - INPUT K, Y Z

21. Write an algorithm as well as draw a flowchart for the following:

Input

- the item ID number
- the Number On Hand
- the Price per item
- the Weight per item in kg
- the Number Ordered
- the Shipping Zone (1 letter, indicating the distance to the purchaser)

Processing

The program will read each line from the user and calculate the following:

Total Weight = Weight Per Item * Number Ordered

Weight Cost = $3.40 + \text{Total Weight} / 5.0$

Shipping cost is calculated as follows:

If Shipping Zone is 'A'

Then Shipping Cost is 3.00

If Shipping Zone is 'B'

Then Shipping Cost = 5.50

If Shipping Zone is 'C'

Then Shipping Cost = 8.75

Otherwise Shipping Cost is 12.60

Handling Charges = 4.00, a constant

New Number On Hand = Number On Hand - Number Ordered

Discount is calculated as follows:

If New Number On Hand < 0

Then Discount = 5.00

Else Discount = 0

Here the purchaser is being given a discount if the item has to be repeat ordered. Total cost is calculated as follows:

Total Cost

= Price of Each * Number Ordered +
Handling Charge + Weight Cost +
Shipping Cost - Discount

For each purchase, print out the information about the purchase in a format approximately like this:

Item Number: 345612
Number Ordered: 1
Number On Hand: 31
Price of Each: 19.95
Weight of Each: 3
Shipping Zone: A
Total Cost: 30.95

After all the purchases are finished, print two lines stating the total number of purchases and the total cost of all purchases.

22. Fill in the blanks.

- A program flowchart indicates the _____ to be performed and the _____ in which they occur.
- A program flowchart is generally read from _____ to _____.
- Flowcharting symbols are connected together by means of _____.
- A decision symbol may be used in determining the _____ or _____ of two data items.
- _____ are used to join remote portions of a flowchart.
- _____ connectors are used when a flowchart ends on one page and begins again on another page.
- A _____ symbol is used at the beginning and end of a flowchart.
- The flowchart is one of the best ways of _____ a program.
- To construct a flowchart, one must adhere to prescribed symbols provided by the _____.
- The programmer uses a _____ to aid him in drawing flowchart symbols.

23. Define a flowchart. What is its use?

24. Are there any limitations of a flowchart?

25. Draw a flowchart to read a number given in units of length and print out the area of a circle of that radius. Assume that the value of pi is 3.14159. The output should take the form: The area of a circle of radius _____ units is _____ units.

26. Draw a flowchart to read a number N and print all its divisors.

27. Draw a flowchart for computing the sum of the digits of any given number.

28. Draw a flowchart to find the sum of N odd numbers given.

29. Draw a flowchart to compute the sum of squares of integers from 1 to 50.
30. Write a program to read two integers with the following significance.
The first integer value represents a time of day on a 24-hour clock, so that 1245 represents quarter to one mid-day.

The second integer represents a time duration in a similar way, so that 345 represents three hours and 45 minutes.
This duration is to be added to the first time and the result printed out in the same notation, in this case 1630 which is the time 3 hours and 45 minutes after 1245.
Typical output might be start time is 1415. Duration is 50. End time is 1505.

CASE STUDY

Problem Statement

Write an algorithm to compute and print the sum of the following series:

$$x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

Analysis

From the problem statement, it is evident that the value of x and the number of terms to be summed up should be taken as input and the sum has to be printed.

Analyzing the expression for the above series, it is seen that the powers and the factorials vary in the sequence 1, 3, 5, 7, ...

Thus,

$$\frac{x}{1!}$$

$$\frac{x^3}{3!} = \frac{x \cdot x \cdot x}{3 \cdot 2 \cdot 1} = \frac{x}{1!} \cdot \frac{x^2}{3 \cdot 2}$$

$$\frac{x^5}{5!} = \frac{x \cdot x \cdot x \cdot x \cdot x}{5 \cdot 4 \cdot 3 \cdot 2 \cdot 1} = \frac{x^3}{3!} \cdot \frac{x^2}{5 \cdot 4}$$

$$\frac{x^7}{7!} = \frac{x \cdot x \cdot x \cdot x \cdot x \cdot x \cdot x}{7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1} = \frac{x^5}{5!} \cdot \frac{x^2}{7 \cdot 6}$$

and so on.

Therefore each term in the given series can be described as $T_k = T_{k-1} \times t$,

where T_k is the k^{th} term and T_{k-1} is the $(k-1)^{\text{th}}$ term, while the variable t for each of the terms are:

$$\frac{x^2}{3 \cdot 2}, \frac{x^2}{5 \cdot 4}, \frac{x^2}{7 \cdot 6}, \frac{x^2}{9 \cdot 8}, \dots$$

respectively. So t can be described by the general form

$$\frac{x^2}{i \cdot (i-1)} \text{ for } i = 3, 5, 7, 9, \dots$$

The following expression can be used repetitively to generate the positive and negative sign for the alternative terms:

$$\text{sign} = -1 \times \text{sign}$$

The initial value of k should be 1. At each iteration, 2 is added to k so that the values of k is generated as 3, 5, 7, and so on. For each iteration, the term is given by the following statement.

$$T = (-1) * T * x * x / (i * (i - 1))$$

The initial value of T is x . The sum of terms should be calculated by the statement $S = S + T$.

The initial value of S is 0.

Having evolved the above expressions, the following statements should be repeated for N times, where N is the number of terms to be summed up to give the final sum of the series.

```
S = S + T
i = i + 2
T = (1) * T * x * x / (i * (i - 1))
```

The number of iterations can be controlled by using a counter variable c . It may be initialized to 1 and the iterations should continue for the values 1, 2, 3, 4, ... N .

Here i can be used to control the iteration. The value of i varies in the sequence 1, 3, 5, 7, ... It is therefore clear that to repeat the iteration twice, the values of i should be 1 and 3. To iterate thrice, the values of i should be 1, 3 and 5. To repeat the statements four times, the values of i should be 1, 3, 5, and 7. Thus it is obvious that the final value of i is just one short of the twice the number of repetitions. Therefore, the condition for which iteration should continue is given by the expression $i < N * 2$. Finally the algorithm is created as shown below.

Algorithm

1. START
2. PRINT "ENTER THE VALUE OF X"
3. INPUT X
4. PRINT "HOW MANY TERMS?"
5. INPUT N
6. I ← 1
7. T ← X
8. S ← 0
9. S ← S + T
10. I ← I + 2
11. T ← (-1)*T*X*X/(I*(I-1))
12. IF I < N*2 THEN GOTO 9
13. PRINT S
14. STOP