

1

Introduction to Computers

1.1 INTRODUCTION

A computer is a device capable of performing computations and making logical decisions at speeds of millions and even billions of times faster than human beings can. For example, many of today's personal computers can perform hundreds of millions of arithmetic and logical operations per second. A person operating a desk calculator might require decades to complete the same number of calculations that a powerful personal computer can perform in one second. Today's fastest *supercomputers* can perform hundreds of billions of additions per second – about as many calculations as hundreds of thousands of people could perform in one year! Moreover, trillion instruction-per-second computers are already in use in research laboratories!

Computers process **data** under the control of sets of instructions called **computer programs**. These programs guide the computer through orderly sets of actions specified by people called **computer programmers**.

1.2 COMPUTER SYSTEMS

A computer comprises of various devices such as keyboard, screen, mouse, disks, memory, CD-ROM and processing units that are referred to as **hardware**. The computer programs that run on a computer are referred to as **software**. Hardware costs have been declining dramatically

in recent years, to the point that personal computers have been rising steadily as programmers develop more powerful and complex applications.

1.2.1 Hardware

Regardless of differences in physical appearances, virtually every computer may be divided into six logical units as shown in the Fig. 1.1.

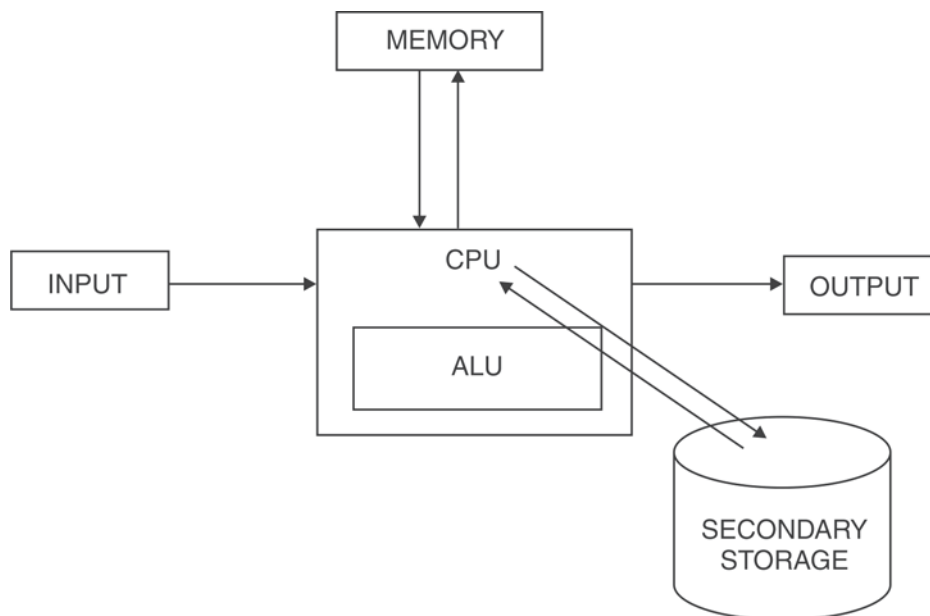


Fig.1.1 Block diagram of computer system

1. Input unit

This is the receiving section of the computer. It obtains information (data and computer programs) from various input devices and places this information at the disposal of the other units so that the information may be processed. Most information is entered into computers today through keyboards and mouse devices. Information can also be entered by speaking to your computer and by scanning images.

2. Output unit

This section of the computer takes information that has been processed by the computer and places it on various output devices to make the information available for use outside the computer. Most information output from computers today is displayed on screens, printed on paper, or used to control other devices.

3. Memory unit

It retains information that has been entered through the input unit so that it may be made immediately available for processing when it is needed. The memory unit also retains processed information until that information can be placed on output devices by the output unit. The memory unit is often called either **Memory** or **Primary Memory**.

4. Arithmetic and logic unit

This section of computer is responsible for performing calculations such as addition, subtraction, multiplication and division. It contains decision mechanisms that allow the computer to complete tasks such as comparing two items from the memory unit to determine whether or not they are equal. Usually ALU is a part of CPU.

5. Central Processing Unit (CPU)

This section of the computer is responsible for supervising the operation of the other sections. The CPU tells the input unit when information should be read into the memory unit, tells the arithmetic and logic unit (ALU) when information from the memory unit should be used in calculations and tells the output unit when to send information from the memory unit to certain output devices.

6. Secondary storage unit

This is the long term, high capacity storage section of the computer. Programs or data not actively being used by the other units are normally placed on secondary devices (such as disks) until they are again needed, possibly hours, days, months or even years later. Information in secondary storage takes much longer to access than information in primary memory. The cost per unit secondary storage is much less than the cost per unit of primary memory.

1.2.2 Software

A set of instructions to the computer (or physical components of the computer) is called programs or software. These sets of instruction or programs can be mainly divided into **System Software** (Operating System) and **Application Software**.

1. System Software (Operating System)

Early computer were capable of performing only on **job** or **task** at a time. This form of computer operation is often called single-user **batch processing**. The computer runs a single program at a time while processing data in groups or **batches**. In these early systems, users generally submitted their jobs to a computer center on decks of punched cards. Users often had to wait hours or even days before printouts were returned to their desks.

Software systems called **operating systems** were developed to help make it more convenient to use computers. Early operating systems managed to transition between jobs. This minimized the time it took for the computer operators to switch between jobs and hence increased the amount of work or **throughput**, computers could process.

As computers became more powerful, it became evident that single user batch processing rarely utilized the computer's resources efficiently because most of the time was spent waiting for slow input/output devices to complete their tasks. Instead, it was thought that many jobs or tasks could be made to **share** the resources of the computer to achieve better utilization. This is called **multiprogramming**. Multiprogramming involves the **simultaneous** operation of many jobs on the computer – the computer shares its resources among the jobs competing for its attention. With early multiprogramming operating systems, users still submitted jobs on decks of punched cards and waited hours or days for results.

In the 1960s, several groups in industry and the universities pioneered **timesharing** operating systems. Timesharing is a special case of multiprogramming in which users access the computer through **terminals**, typically devices with keyboards and screens, sharing the computer at once. The computer does not actually run jobs of all the users simultaneously. Rather, it runs a small portion of one user's job and then moves on to service the next user. The computer does this so quickly that it may provide service to each user several times per second. Thus the users programs **appear** to be running simultaneously. An advantage of timesharing is that the user receives almost immediate responses to requests rather than having to wait long periods for results as with previous modes of computing.

2. Application Software

Programs or set of instruction to the computer which will assist the user in performing specific tasks are called *Application Software*. Some examples of application software are MS Office, Tally, Oracle, and Adobe Photoshop. MS Office is used for creating documents, spread sheets, presentations, database creation etc... Tally is used for accounting purposes in business applications. Oracle is used for database creation and maintenance. Adobe Photoshop is used for photo editing and creation.

1.3 COMPUTING ENVIRONMENTS

Computers can be used in different environments. An environment describes a situation. There are basically three types of computing environments depending on the way the computer are used. They are:

- Personal Computing
- Distributed Computing
- Client/Server Computing

1.3.1 Personal Computing

In 1977, Apple Computer popularized the concept of **personal computing**. Initially, it was a hobbyist's dream. Computers became economical enough for people to buy them for their own personal or business use. In 1981, IBM, the world's largest computer vendor, introduced the IBM Personal Computer. Literally overnight, personal computing became legitimate in business, industry and government organizations.

1.3.2 Distributed Computing

Computers were **stand-alone** units – people did their work on their own machines and then transported disks back and forth to share information (this is often called **sneaker net**). Although early personal computers were not powerful enough to timeshare several users, these machines could be linked together in computer networks, sometimes over telephone lines and sometimes in **local area networks (LANs)** within an organization. This led to the concept of **distributed computing**, in which an organization's computing, instead of being performed strictly at a central computer installation, is distributed computers were powerful enough to handle the computing requirements of individual users and to handle the basic communications tasks of passing information back and forth electronically.

1.3.3 Client/Server Computing

Today's most powerful personal computers are as powerful as the million dollar machines of just a decade ago. The most powerful desktop machines – called **workstations** – provide individual users with enormous capabilities. Information is easily shared across computer networks, where computers called **file servers** offer a common store of programs and data that may be used by client computers distributed throughout the networks, hence the term **client/server computing**. C and C++ have become the programming languages of choice for writing software for operating systems, which include **UNIX, LINUX AND MICROSOFT WINDOWS** – based systems.

1.4 COMPUTER LANGUAGES

Programmers write instructions in various programming languages, some directly understandable by the computer and others that require intermediate **translation** steps. Hundreds of computer languages are in use today. These may be divided into three general types:

1. Machine Languages
2. Assembly Languages
3. High-level Languages

1.4.1 Machine Languages

Any computer can directly understand only its own **machine language**; machine language is the **natural language** of a particular computer. It is defined by the hardware design of the computer. Machine languages generally consist of strings of numbers (ultimately reduced to 1s and 0s) that instruct computers to perform their most elementary operations one at a time. Machine languages are **machine dependent**, i.e., a particular machine language can be used on only one type of computer. Machine languages are cumbersome for humans and therefore can not be easily used for programming. For example, a machine level program to add allowance to basic pay could comprises of series of instructions with 0s and 1s.

1.4.2 Assembly Languages

As computers became more popular; it became apparent that machine language programming was too slow, tedious and error prone. Instead of using strings of numbers that computers could directly understand, programmers began using English-like abbreviations formed the basis of **assembly languages**. **Translators programs** called **assemblers** were developed to convert assembly language programs to machine language at computer speeds. The following section of an assembly language program also adds allowance to basic pay and stores the result in gross pay, but more clearly than is done in machine language.

```
LOAD BASEPAY  
ADD ALLOWANCE  
STORE GROSSPAY
```

Although such code is clearer to humans, it is incomprehensible to computers until translated to machine language.

1.4.3 High-level Languages

Computer usage increased rapidly with the advent of assembly languages, but these still required many instructions to accomplish even the simplest tasks. To speed the programming process, **high-level languages**, in which single statements accomplish substantial tasks, were developed. Translator programs called **compilers** convert high-level language programs into machine language. High-level languages allow programmers to write instructions that look almost like everyday English and contain commonly used mathematical notations. A payroll program written in a high level language might contain a statement such as:

```
grosspay=basepay+allowance
```

Obviously, high-level languages are much more desirable from the programmer's stand point than either machine languages or assembly languages. C and C++ are among the most powerful and most widely used high-level languages.

1.5 CREATING AND RUNNING PROGRAMS

The process of compiling a high-level language program into machine language can take a considerable amount of computer time. This problem was solved by the development of **interpreter** programs that can directly execute high-level language programs without needing to compile them into machine language. Although compiled programs execute faster than interpreted programs, interpreters are popular in program development environments in which programs are changed frequently as new features are added and errors are corrected. Once a program is developed, a compiled version can be produced to run most efficiently.

What is an Interpreter?

An **interpreter** reads an executable source program written in high level programming language as well as data for this program, and it runs the program against the data to produce some results. Interpreter executes one instruction at a time as you enter the instruction.

Eg. Unix Shell Interpreter, which runs operating system commands interactively and Visual Basic Interpreter.

What is a Compiler?

A **compiler** is a program that translates a source program written in some high level programming language (such as C) into machine code for some computer architecture (such as the Intel Pentium architecture). The generated machine code can be later executed many times against different data each time.

Eg. Turbo 'C' Compiler and Borland 'C' Compiler.

1.6 SOFTWARE DEVELOPMENT STEPS

Before we go further it is better we look at a systematic program development and problem solving. In order to accomplish computerizing the solution to a problem the following six steps are needed:

1. *Specifying requirements:* The problem whose solution is to be implemented on a computer through a program should be thoroughly specified and understood.
2. *Analysis:* The problem must be analyzed to determine the inputs and outputs needed.
3. *Designing algorithm:* A solution must be conceived and must be represented step by step by using algorithmic/pseudo code notations or flow chart symbols. This will help thoroughly verifying the correctness of the solution to the problem.
4. *Implementation:* The flow charts and algorithms developed in the previous steps are converted into actual programs in the high level languages like C. Next translate the program in high level language into machine code. This process is known as **Compilation**.

Syntactic errors are found quickly at the time of compiling the program. These errors occur due to the usage of wrong syntaxes for the statements.

Eg. $x=a*y+b$

There is a syntax error in this statement, since, each and every statement in 'C' language ends with a semicolon(;).

Most of high level language compiler implementations will generate diagnostic messages when syntax errors are detected during compilation.

5. *Testing*: This deals with the proper and correct execution of the program. The program is executed with input data. In this phase, we may encounter two types of errors.

Runtime errors:

These errors may occur during the execution of programs even though the program is successfully compiled without syntax errors. The most common types of runtime errors are:

- Eg.
1. Array range out of bound
 2. Divided by zero

Logical errors:

These errors occur due to incorrect usage of the instructions in the program. These errors are neither detected during compilation or execution nor cause any stoppage to the program execution. They only produce incorrect outputs. When the program ends up with incorrect outputs the logical errors are to be identified and rectified.

6. *Maintenance and updation*: After the software is delivered to the customer and installed at the premises of the customer, the customer may ask for some changes after using the software for some time. This calls for updation of the software with some changes.

This approach makes us to visualize the logic involved in the solution to the problem and further allows you to complete the task of program writing and successful execution.

1.7 ALGORITHM/PSEUDOCODE

Pseudocode is an artificial and informal language that helps programmers develop algorithms. An algorithm is a step-by-step procedure for solving a problem using a pseudocode. The pseudocode we present here is particularly useful for developing algorithms that will be converted to structured C programs.

Pseudocode is similar to everyday English; it is convenient and user-friendly although it is not an actual computer programming language. Pseudocode programs are not actually executed on computers. Rather, they merely help the programmer "*think out*" a program before attempting to write it in a programming language such as C.

Example 1. An Algorithm / pseudo code to add two numbers.

Step 1 : Start
Step 2 : Read the two numbers into a, b
Step 3 : $c = a + b$
Step 4 : Write/print c
Step 5 : Stop.

Example 2. An Algorithm/ Pseudo code to find whether a given number is odd number or a even number.

Step 1 : Start
Step 2 : Read the number n
Step 3 : If $(n \% 2) = 0$ (i.e. the remainder is zero) then.
Write 'n is even number' Go to step 5
Step 4 : Write 'n is odd number'
Step 5 : Stop.








Example 3. An algorithm/pseudocode to read three numbers and to determine the maximum, second highest and the minimum.

Step 1 : Read the three numbers into a, b, c
Step 2 : If $a > b$ and $a > c$
Max = a;
If $b > c$
Max2 = b, Min = c
Else
Max2 = c, Min = b
Goto Step 5
Else
Goto Step 3
Step 3 : If $b > c$ then do the following else goto Step 4
Max = b;
If($a > c$)
Max2 = a, Min = c
Else
Max2 = c, Min = b;
Goto Step 5
Step 4 : Max = c;
If ($a > b$)
Max2 = a, Min = b
Else
Max2 = b, Min = a
Goto Step 5.
Step 5 : Print Min, Max2, Max;
Step 6 : Stop.

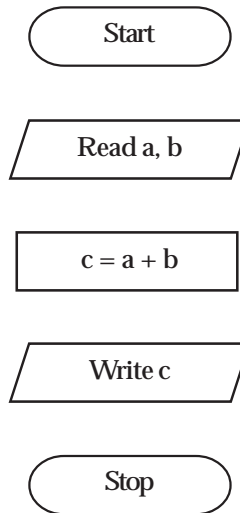
1.8 FLOW CHART

A flowchart is a graphical representation of an algorithm or a portion of an algorithm. Flowcharts are drawn using certain special-purpose symbols such as rectangles, diamonds, ovals and small circles as shown in Table 1.1. These symbols are connected by arrows called flowlines. Like pseudocode, flowcharts are useful for developing and representing algorithms, although, pseudocode is preferred by most programmers. Flowcharts clearly visually show how control structures operate in a program. The most common symbols used in drawing flow charts are in Table 1.1.

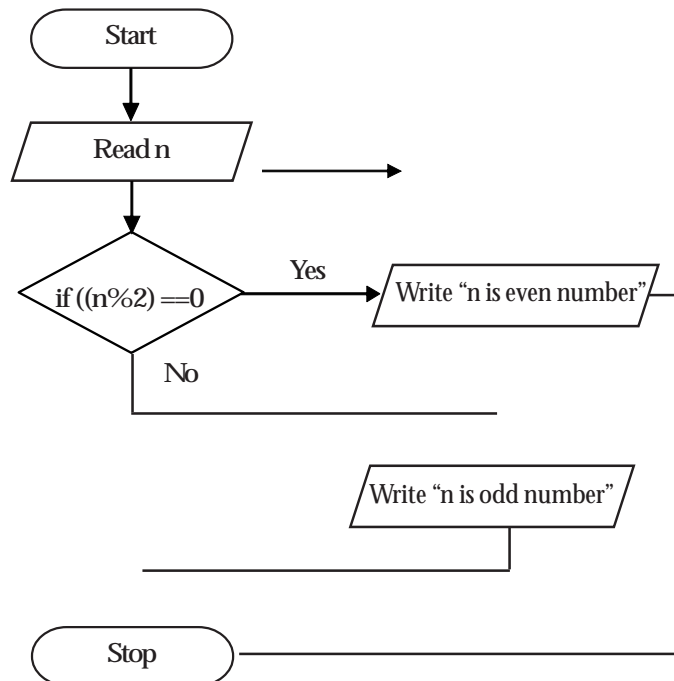
Table 1.1 Flow Chart Symbols

Oval		Terminal	Start/Stop/Begin/End.
Parallelogram		Input/Output	Making Data available for processing (Input) or recording of the processed Information (Output).
Document		Print Out	Show Data Output in the form of Document.
Rectangle		Process	Any Processing to be done. A Process changes or moves Data. An Assignment Operation.
Diamond		Decision	Decision or Switching type of Operations.
Circle		Connector	Used to connect different parts of Flowchart.
Arrow		Flow	Joins two symbols and also represents flow of Execution.

Example 1. Flowchart for addition of two numbers.



Example 2. Flowchart to find whether a given number is odd or even.



1.9 SOFTWARE DEVELOPMENT LIFE CYCLE

The Systems Development Life Cycle (SDLC) or Software Development Life Cycle in software engineering is the process of creating or altering systems (softwares) and the models and methodologies that people use to develop these systems (softwares).

Different phases of SDLC are as follows:

1. Initiation/Planning and requirements collection
2. Analysis
3. Design
4. Build or coding
5. Testing
6. Maintenance and updation

These steps are similar to what we have discussed in earlier section. In software engineering the SDLC concept supports many kinds of software development methodologies (models). These methodologies form the framework for planning and controlling the creation of an information system. One such model, which is frequently used and very much similar to SDLC is Water Fall Model.

Water fall model is a sequential software development model in which progress is seen as flowing steadily downwards (like a waterfall) through the phases of initiation or planning and requirements collection, analysis, design, build or coding, testing and maintenance and updation. The phases SDLC in water fall model are represented in the Fig. 1.2.

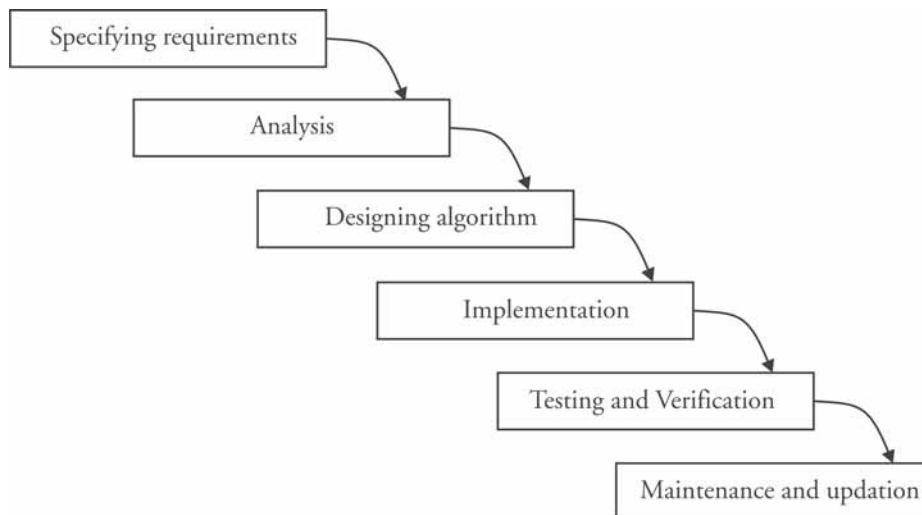


Fig 1.2 Representation of SDLC phases in water fall model

1.10 APPLICATION OF SOFTWARE DEVELOPMENT METHOD

An example problem is presented in this section to illustrate how to apply software development method. After the problem statement in the analysis we identify the data requirements of the

problem including the inputs and desired outputs. Next an algorithm is designed and refined to solve the problem.

Finally we implement the algorithm as a program written in C language. We also indicate how to test the program. Though the problem being taken is a trivial one, the student is urged to observe the process and adopt similar steps in solving other problems.

Problem:

You would like design a program that converts temperature in Fahrenheit to temperature in centigrade.

Analysis:

You should be very clear about the problem before you try to solve it. In this problem we are asked to convert the measurement of temperature from one system to another. You should be clear that the convention is from Fahrenheit to centigrade and not vice-a-versa. Therefore the problem input is temperature in $^{\circ}\text{F}$ and the problem output is temperature in $^{\circ}\text{C}$. The data requirements and the conversion formula are listed below:

Data requirements

- Input: temperature in $^{\circ}\text{F}(f)$
- Output: temperature in $^{\circ}\text{C}(c)$
- Conversion formula: $c = ((f - 32) \times 5) \div 9$

Design:

We now need to formulate the algorithm (step by step procedure)

Algorithm:

Begin

- Step 1: Read the temperature in $^{\circ}\text{F}$
- Step 2: Convert the temperature into $^{\circ}\text{C}$
- Step 3: Display the temperature in $^{\circ}\text{C}$

End

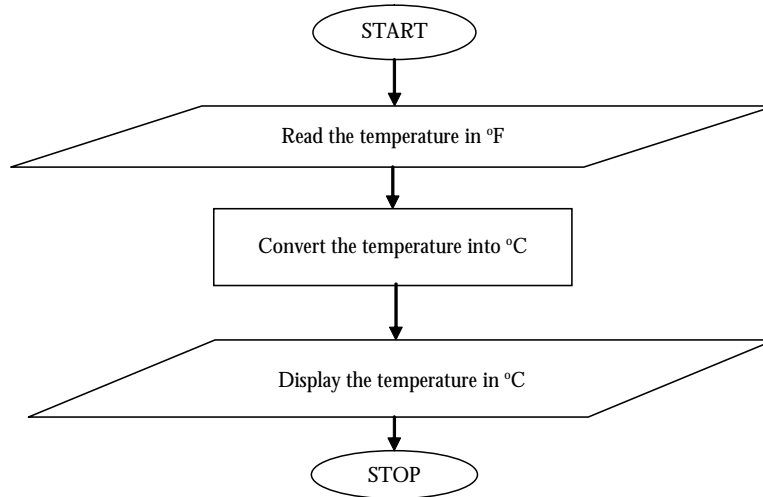
The above is the 1st phase of the design. We observe that Step 2 can be further subdivided into

- 2.1 Subtract 32 from $^{\circ}\text{F}$
- 2.2 Multiply the result of Step 2.1 by 5
- 2.3 Divide the result of Step 2.2 by 9

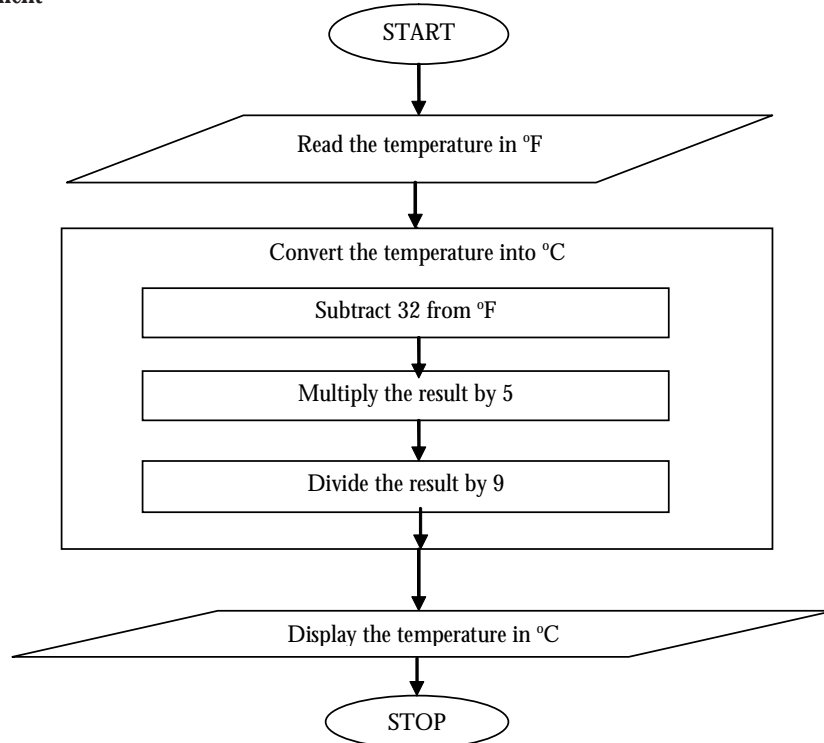
Thus incorporating the method of conversion from $^{\circ}\text{F}$ to $^{\circ}\text{C}$.

Flow Chart:

Phase 1:

*Fig.1.3 Temperature conversion flow chart*

After Refinement

*Fig.1.4 Refined temperature conversion flowchart*

Implementation:

The next step is implementation where we convert the algorithm/flow chart into a C program.

```
1. #include<stdio.h>
2. void main()
3. {
4. float f,c;
5. printf("Enter the temperature in Fahrenheit:");
6. scanf("%f", &f);
7. c=f-32;
8. c=c*5;
9. c=c/9;
10. printf("Celsius=%f\n",c);
11. }
```

Output:

```
Enter the temperature in Fahrenheit:60
Celsius=15.555555
```

Fig. 1.5 Sample program with, sample execution results

Fig. 1.5. shows the C program with sample execution results. However the above c program can be refined as the three steps at lines 7 to 9 can be combined to one line since the conversion formula can be directly coded as an expression. Fig. 1.6. shows the revised.

```
1. #include<stdio.h>
2. void main()
3. {
4. float f,c;
5. printf("Enter the temperature in Fahrenheit:");
6. scanf("%f", &f);
7. c=((f-32)*5)/9;
8. printf("Celsius=%f\n",c);
9. }
```

Fig. 1.6 Refined sample program

Testing:

To verify that the program works properly, enter a few more values of temperature in °F and verify whether you are getting the correct results or not. You may try some negative temperature also. Typical test cases are to be designed for more complex programs to unearth any runtime or logical errors that may occur.

Maintain and update:

This deals with the proper maintenance of the software after delivering it to the customer. And if there are any requirements that are found while maintaining the software they can be programmed and updation of the software is done.

This is how the software development method can be applied to any problem to solve it using computers (computer programming language (high-level language)).

PROBLEMS & EXERCISES

1. Write algorithm/pseudo code and flow charts for the following problems.
 - (a) Finding largest number in two numbers.
 - (b) Roots of quadratic equation.
 - (c) To generate all even numbers between two given numbers.
 - (d) Finding the average of n numbers.