--------- **Eighth Edition** ----------

PROGRAMMING

IN

ANSI C

E. BALAGURUSAMY

PROGRAMMING IN ANSI C

Eighth Edition

About the Author

E Balagurusamy is presently the Chairman of EBG Foundation, Coimbatore. In the past he has also held the positions of member, Union Public Service Commission, New Delhi and Vice-Chancellor, Anna University, Chennai. He is a teacher, trainer and consultant in the fields of Information Technology and Management. He holds an ME (Hons) in Electrical Engineering and PhD in Systems Engineering from the Indian Institute of Technology, Roorkee. His areas of interest include Object-Oriented Software Engineering, E-Governance: Technology Management, Business Process Re-engineering and Total Quality Management.

A prolific writer, he has authored a large number of research papers and several books. His best-selling books, among others include:

- *Fundamentals of Computers*
- *Computing Fundamentals and C Programming 2/e*
- *Programming in C#, 4/e*
- *Programming with Java, 5/e*
- *Object-Oriented Programming with C++, 7/e*
- *Programming in BASIC, 3/e*
- *Numerical Methods*
- *Reliability Engineering*

A recipient of numerous honors and awards, he has been listed in the Directory of Who's Who of Intellectuals and in the Directory of Distinguished Leaders in Education.

PROGRAMMING IN ANSI C

Eighth Edition

E Balagurusamy

Chairman
EBG Foundation
Coimbatore

McGraw Hill Education (India) Private Limited

CHENNAI

Senior Manager—Production: *Piyaray Pandita*

General Manager—Production: *Rajender P Ghansela*
Manager—Production: *Reji Kumar*

Preface

C is a powerful, flexible, portable and elegantly structured programming language. Because C combines the features of high-level language with the elements of an assembler, it is suitable for both systems and applications programming. It is undoubtedly the most widely used general-purpose language today in operating systems, and embedded system development. Its influence is evident in almost all modern programming languages. Since its standardization in 1989, C has undergone a series of changes and improvements in order to enhance the usefulness of the language. The version that incorporates the new features is now referred to as C11.

New to this Edition

Distinguished as one of the bestsellers in the market, 'Programming in ANSI C' is taking forward another step in the pursuit of perfection. The title keeps up with its predecessors by balancing between theory and practical approach of the subject. In order to build a firm foundation, a new chapter 'Introduction to Computing' has been introduced. The appendix 'Projects' has been revamped with two new and exciting projects, that will help students to hone their programming skills and bring them to practical usefulness. Existing chapters have been further enriched with additional write-ups, tips and a new 'Test Your Skills' section around the content. Two new sections, Multiple Choice Questions and Interview Questions, have been introduced in each chapter, with the intent to give more practice to students and prepare them for different interviews and competitive exams. The section on Programming Exercises has also been enriched with some new questions.

Organization of the Book

Programming in ANSI C in its eighth edition starts with two introductory chapters. Chapter 1, 'Introduction to Computing', introduces students to different components of a computer and discusses problem solving aspects of computing. Chapter 2, 'Overview of C', discusses the basic structure of C programs and their execution. Chapter 3 discusses how to declare the constants, variables, and data types. Operators and expressions are presented in Chapter 4. Chapter 5 deals with the managing of Input and Output Operations. Chapter 6 talks about Branching. The concept of decision making and looping is explained in Chapter 7. Arrays, Character Arrays, and Strings have been discussed in Chapters 8 and 9. User-defined functions, structures and unions are covered in Chapters 10 and 11, while Chapter 12 talks about Pointers. Chapter 13 briefs about file management in C. Dynamic Memory Allocation and Linked Lists are presented in Chapter 14. Preprocessor is explained in Chapter 15. Guidelines helpful in developing a C Program are listed in Chapter 16.

Salient Features of the Book

- A comprehensive book covering the fundamentals as well as latest features of C programming.

- Each chapter follows an outcome-based learning approach as per 'Bloom's Taxonomy'.

- Strong pedagogical features, that include:
  - ♦ Worked out problems
  - ♦ Tips and important notes
  - ♦ 24 detailed case studies
  - ♦ Multiple choice questions
  - ♦ Programming and debugging questions
  - ♦ Interview questions

- Two new and exciting projects on:
  - ♦ Electricity Board Management System
  - ♦ Making Web Services in C

- An engaging appendix on 'Graphic Programming Using C'

Digital Supplements

The digital supplement can be accessed at the given link (http://www.mhhe.com/balagurusamy/ansic8e). It contains the following components:

- Projects
- e-Case studies
- Additional information and questions

Acknowledgements

This book is my sincere attempt to make a footprint on the immensely vast and infinite sands of knowledge. I would request the readers to utilise this book to the maximum extent.

I owe a special thanks to the entire team of McGraw Hill Education India.

E Balagurusamy

Publisher's Note

McGraw Hill Education (India) invites suggestions and comments from you, all of which can be sent to *info.india@mheducation.com* (kindly mention the title and author name in the subject line).

Piracy-related issues may also be reported.

Contents

Chapter 3 Constants, Variables and Data Types

Chapter 10   User-Defined Functions

Chapter 13   File Management in C

Introduction to Computing

INTRODUCTION

Computing is an operation that involves processing of data by making use of a computer. To be able to use a computer and perform computation, we first need to be aware of the different components that are present in it and understand how these components may be used to serve our purposes. In this chapter, we will briefly discuss the different hardware components of a computer. We will also discuss methods to develop programs using which we can achieve our objective to solve problems and perform various computational activities.

COMPONENTS OF A COMPUTER

A computer is an electronic device which may be used to perform various computations involving arithmetic and logical operations. Figure 1.1 shows the major components of a computer.

LO 1.1
Know the structure of a computer and have an idea about its functioning
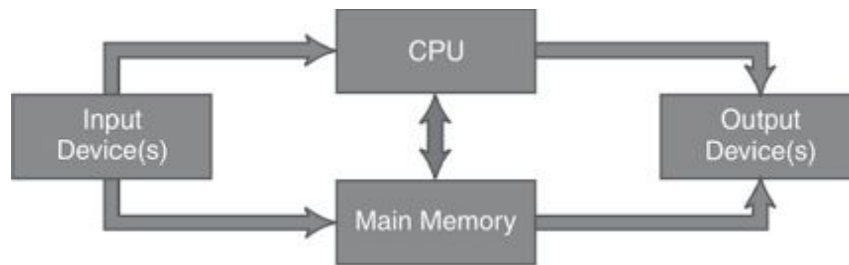
Fig. 1.1 Components of a computer

As seen in the figure, the major components in a computer are the central processing unit (CPU), input device(s), output devices(s) and the memory. These components interact with each other using buses, as shown in the figure.

Central Processing Unit

The CPU is the unit in which all processing activities take place. The CPU is a classic example of a very large scale integrated (VLSI) electronic circuit that consists of the arithmetic and logic unit (ALU), the control unit (CU) and some registers. Figure 1.2 shows the layout of a CPU. The ALU contains circuits using which various arithmetic and logical operations may be executed. These circuits include adders (for addition), subtractors (for subtraction), comparators (for comparison of operands), etc. The CU, also known as the brain of a computer, is the unit which is responsible for generating control signals. These control signals control the working of the various components within a computer. The registers present inside the CPU may be considered to be a set of high-speed storage devices meant to serve as working memory for the ALU. Some of these registers are assigned the task of holding special data or instructions. Such registers are known as special purpose registers, whereas others are known as general purpose registers. The units present inside the CPU interact with each other through an internal bus. Signals generated by the CU also control the activities of the various components within the CPU. Apart from the internal bus, three other buses also interface the CPU. The control bus carries control signals generated by the CU to the memory and all the peripherals. The address bus carries an address to locate a word in the memory. Finally, the data bus is

used to carry data and information to and from between the components of the computer. In addition to the units mentioned so far, modern day CPUs are also equipped with high-speed, on-chip memory units known as the cache memory.
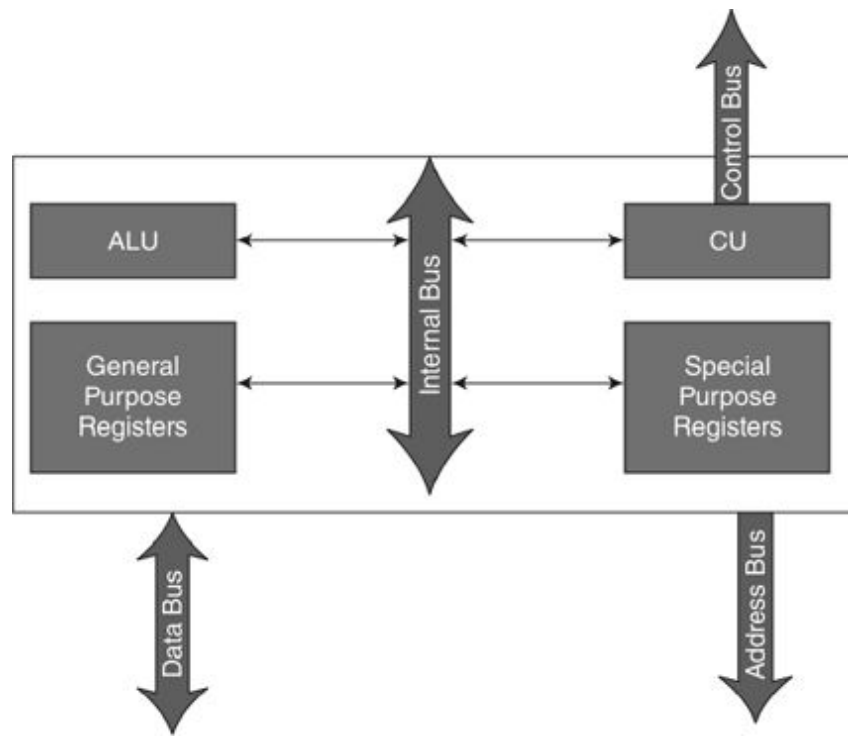
Fig. 1.2 Layout of a model CPU

Input Devices

Input devices are the ones through which users are allowed to input data and other instructions to the computer. The most popular choices for input devices are the keyboard and the mouse. Other examples include scanners, readers, etc. Most input devices are electro-mechanical in nature which accept the inputs from the user through mechanical means, such as depressing of keys in a keyboard to enter some data or moving the mouse around on a surface to locate a position. Information acquired through such mechanical means is digitized (converted into electronic signals) and transmitted to the CPU/memory.

Output Devices

Just as input devices are used to convey messages to the computer, output devices are used by the computer to convey messages back to the users. The most common output device that almost all computers have is the monitor. The monitor is used to display the results of computations, messages, etc. Other examples of output devices are printers, speakers, plotters, etc.

The input and the output devices are sometimes known as peripheral devices. These auxiliary devices may be connected to a computer and used. They are auxiliary because their existence within a computer is not mandatory unlike the CPU or the memory.

Memory

Memory/storage unit is the component of a computer that serves data storage purposes. These units are of varying access speeds and volatility. Access speed refers to the time taken to store/retrieve data from the memory. Volatility refers to the duration of time or the conditions under which data is stored in the memory. Based on these characteristic features, the computer's memory may be categorized into different types. Figure 1.3
shows the classification of the computer's memory.
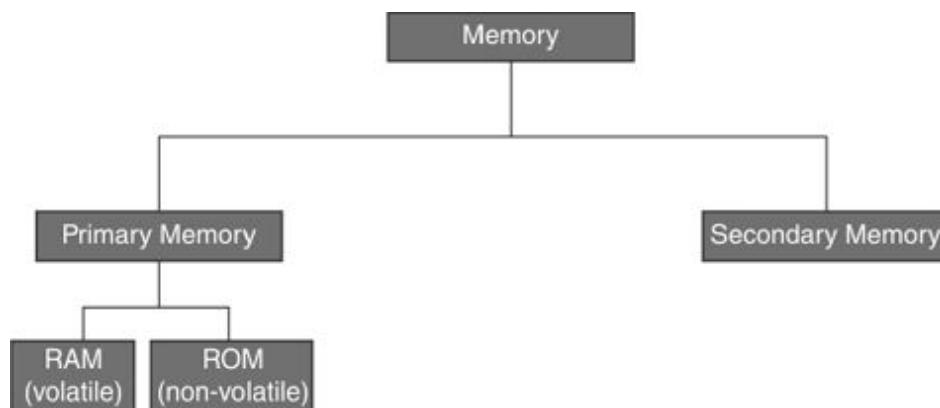
Fig. 1.3 Classification of the computer's memory

As shown in the figure, the memory is broadly classified into primary memory and secondary memory. The primary memory, also known as the main memory, is so called because the CPU can directly interact with it through the data bus and the address bus. The main memory is of two major types, the random access memory (RAM) and the read only memory (ROM). The RAM is a volatile memory that stores instructions and data related to the program currently being executed. The RAM is volatile because it would lose all the information stored in it as soon as the power to the computer is switched off. Unlike the RAM which allows both the read and write operations, the ROM is a read-only, non-volatile memory. The ROM does not permit any write operation and is non-volatile because the information stored in it would be retained even after the computer has been switched off. Apart from other important routines, the ROM houses the boot strap loader. The boot strap loader is a system software that is responsible for loading the memory resident portion of the operating system from the secondary memory into the RAM once the computer has been switched on and the power on self tests (POST) have been conducted. The secondary memory, also known as auxiliary memory, is a large depository of non-volatile storage space that can store information for as long as we want. In this respect it is worth mentioning that the secondary memory is slower compared to the access time of the primary memory. The hard disk, flash drives, etc. are some examples of secondary memory.

CONCEPT OF HARDWARE AND SOFTWARE

The hardware of a computer consists of various electronic, electro-mechanical and electro-magnetic devices along with the interfacing buses connecting them. In other words, the hardware of the computer consists of the bare machinery. Software, on the other hand, consists of the set of instructions, commands, programs, etc. that can make the bare machinery operate and fulfill our objectives. If the hardware is the body, then the software is the soul of a system. For a computer to operate, both the hardware and the software are indispensable.

Types of Software

Depending upon the type of work a software is required to perform, it can be classified into two categories, system software and application software.

System Software

System softwares are the ones that closely interact with the hardware of a computer. These softwares depend upon the architecture of the underlying computer. They are responsible for controlling the resources of the system and perform other activities related to the system. The different kinds of system softwares are operating system, loader, linker and translators.

Operating System

Operating system is a system software which acts as an interface between the user and the bare hardware. The operating systems currently in use are Linux, Windows 10, UNIX, macOS High Sierra, etc. The major functions of an operating system are:

- Process and processor (CPU) management
- Memory management
- Device management
- File management

To put it simply, the operating system manages all the resources of a computer. It also interprets all the commands that it receives from the user to the underlying hardware. Today's operating systems come with a smart graphic user interface (GUI) that is easy to operate. They also offer several utilities such as calculators, text editors, scanning softwares, drawing tools and image editing tools.

Translators

Translators cover a class of system softwares which translate a program written in one programming language into another. Compiler, assembler, interpreter, etc. are few of the most used translators. The source programming language is either a high-level programming language or assembly language. The target language in almost all cases is machine language which is essentially in binary (1's and 0's – high voltage and low voltage), a language understood by the computer. Translators are system softwares because they need a thorough understanding of the architectural configuration of the underlying hardware. This knowledge is necessary because the target code is machine dependent. The machine instruction set supported by a particular architecture may not be supported by another.

Compilers are system softwares that translate a program written in high level language (HLL) into machine language. C, C++, Java, etc. are examples of HLLs. Thus, compilers have to be aware of the syntactic nuances of the HLL and the instruction set, addressing modes, register configuration, etc. of the machine in which the target program is supposed to be executed. The syntactic specification of a programming language is given to the compiler using the grammar of the language. Thus, a compiler designed for the C programming language would not be able to process a program written in the Java language. This is because the C compiler has been provided with the grammar rules of the C language and would consider statements written using Java as incorrect. The output of a compiler is in machine language and uses the instructions supported by the CPU of the computer. Thus, a compiler designed to generate machine code for a computer housing an Intel processor would not be able to generate machine code for a computer containing a Motorola processor. For any compiler it is essential to specify both the source language as well as the target machine in which the output program will be executed. The machine code generated by the compiler is known as object code.

Assemblers translate a program written in assembly language to machine language. Assembly language consists of instructions supported by the underlying processor. The difference between assembly and machine language is that assembly language uses mnemonics such as ADD, SUB,

etc., whereas machine language uses the corresponding binary operation codes for each of these mnemonics.

Linker

The linker links together the object codes belonging to all the sub-routines/functions of a program and forms an executable code. This executable code may be stored in the secondary memory and can be loaded into the main memory as and when required for execution.

Loader

The loader is a system software that loads a program to be executed from the secondary memory into a designated area of the main memory. Modern day operating systems have a loader incorporated within them. Absolute loaders, relocatable loaders, direct linking loaders are few forms of loader. A special loader known as the boot strap loader deserves a special mention in this respect. As mentioned earlier, the boot strap loader loads the memory resident portion of the operating system into the main memory immediately after the computer is switched on. The operating system then takes control of the computer.

Application Software

Application softwares are the ones that are meant to perform a specific activity and aid users in various ways. Word processing softwares, image editing, spread sheet packages, database softwares and accounting softwares are all examples of application softwares. Each of these has been designed to serve a special purpose.

Art of Programming through Algorithms and Flowcharts

We have seen in the previous section how the software is invisible but indispensable part of a computer that enables the users to interact with the computer and makes it work as per one's specification and requirements to achieve a computation objective. A software can be considered as a set of programs which work in a cohesive manner to perform a task. A program is a set of instructions which when executed in sequence performs a task.

Let us consider the example of a program which when executed would enable the user to enter two numbers, add them and display the result. It is clear that this program would contain three sets of instructions; the first set would enable users to input the data, the second set to add the numbers and the third set to display the result. When writing the program, the programmer would have planned these three sets and used statements within it to implement each of them. The example considered here is an extremely simple one.

However, more often than not, the problems which a computer is supposed to solve would be more complex than this. Such complex problems would invariably require complex solutions. These solutions involve a lot of complex arithmetic operations and logical decision making. The need of the hour here is a tool which would enable us to represent the logic required in the program and make it possible to transform the logic to an equivalent set of instructions which when executed by the computer would solve the problem. Algorithms and flowcharts are two such tools. These tools are independent of the programming language as well as the target machine. Thus, the logic represented using one of these tools may be used to write programs in any language of the programmer's choice. In fact, logic should always be independent of programming languages. Irrespective of the tool and language chosen to write a program, programming logic constructs are of three important types (apart from input/output and sub-routine call constructs):

- Imperative statements (action)

- Conditional statements (decision making)

- Iterative statements (repetition/Loops)

The tools which we are talking about should be capable of representing each of these three kinds of constructs because a program comprises statements which fall into one of these three categories.

It is advisable to always use tools such as algorithms and flowcharts to first plan and then write a program. After all it has been rightly said that well planned is half done.

Algorithm

An algorithm is a *finite* set of *unambiguous* instructions which, when executed, performs a task *correctly*. There are three characteristic features in this description of an algorithm. They are:

- The number of steps required to perform the task should be finite.

- Each of the instruction in the algorithm should be unambiguous in nature; meaning on the execution of each such instruction, the outcome should be definite and predictable.

- Finally, the algorithm should solve the problem correctly. In other words, on the execution of the algorithm, the objective should be fulfilled.

There are no defined rules to write an algorithm. However, most authors prefer to use English-like statements to represent the action that needs to be performed at each step. We will consider different examples to demonstrate each of the three logic constructs.

EXAMPLE 1 (Use of imperative constructs)

Problem – Develop an algorithm to find the average of three numbers taken as input from the user.

Algorithm AVERAGE

```
Step 0  START
Step 1  INPUT first number into variable A
Step 2  INPUT second number into variable B
Step 3  INPUT third number into variable C
Step 4  COMPUTE SUM - A + B + C
Step 5  COMPUTE AVG - SUM / 3
Step 6  DISPLAY AVG
Step 7  END
```

Steps 1, 2 and 3 are statements that will enable the input of three different numbers. Step 6 is used to display the result. Steps 4 and 5 are of particular interest to us as these are the two imperative statements which instruct/command to perform the computation. Thus, imperative statements are the ones which command the processor to perform a task.

As mentioned earlier, an algorithm has no set rules for writing. This same algorithm could very well be written by not using the word COMPUTE in steps 4 and 5. In a similar manner, step 1 could have been simply written as 'INPUT A'. Use of words such as COMPUTE in steps 4 and 5, or the phrase, 'first number into variable' in step 1 serves as an annotation, which increases the readability and understandability of the algorithm.

EXAMPLE 2 (Conditional construct)

Problem – Develop an algorithm to divide one number by another and find the quotient.

```
Algorithm DIVISION
Step 0  START
Step 1  INPUT first number into variable A
Step 2  INPUT second number into variable B
Step 3  IF B !- 0 THEN
                Q - A/B
                DISPLAY Q
        END IF
Step 4  END
```

Dividing one number by another is perhaps one of the simplest instructions that can be given to the CPU for processing. However, in dividing two numbers one should always be careful to check whether the divisor is 0. A computer has no way to store an infinitely large number. So, it does not permit a division by 0. An attempt to do so forcefully causes the computer to generate a runtime 'Divide Overflow' error and the processing is immediately aborted. The algorithm developed here checks to see whether the divisor B is 0. Division is performed only if B is not 0 and the result is displayed. Step 3 is the statement where a decision is taken whether the

divisor is 0. We use the word 'IF' to do so. The condition 'B != 0' is evaluated to check whether B is 0. If B is not 0, then the condition is evaluated to be true. In this case, the two statements 'Q = A / B' and 'DISPLAY Q' would be executed in sequence. Once done, the control would move on to step 4 which causes the algorithm to halt. In case B is 0, the condition would evaluate to be false. Then the control of the algorithm would directly move to step 4 without attempting the division operation. The symbol, '!=' corresponds to 'not equal to'. Once again this is not a standard notation for an algorithm. Some authors use '<>' instead to represent 'not equal to'. When transforming into a program, 'not equal to' would be replaced by the corresponding logical operator supported by the programming language. The 'END IF' in step 3 indicates the end of the conditional statement construct.

EXAMPLE 3 (Conditional construct)

Problem – Develop an algorithm to find the maximum of two numbers input by the user.

```
Algorithm MAXIMUM
Step 0  START
Step 1  INPUT first number into variable A
Step 2  INPUT second number into variable B
Step 3  IF A > B THEN
                MAX - A
        ELSE
                MAX - B
        END IF
Step 4  DISPLAY MAX
Step 5  END
```

This example, like the previous one, also demonstrates a conditional statement construct. However, depending upon whether the outcome of the condition 'A > B' is true or false two different tasks would be performed. The use of the word 'ELSE' makes this algorithm slightly different from the previous one. However, decision-making is involved in both of them. The values of A and B are compared to determine whether A is greater than B (A > B). If A is greater than B, then it implies that A is the maximum of the two (MAX = A), else it implies that B is the maximum of the two (MAX = B).

EXAMPLE 4 (Iteration construct)

Problem – Develop an algorithm to find the sum of the first N natural numbers where N would be input by the user.

```
Algorithm SUM-N
Step 0   START
Step 1   INPUT N
Step 2   I - 1
Step 3   SUM - 0
Step 4   REPEAT Steps (a) — (b) WHILE I <-N
                (a) SUM - SUM + I
                (b) I - I + 1
         END WHILE
Step 5   DISPLAY SUM
Step 6   END
```

The objective of this algorithm is to calculate $1 + 2 + \ldots + N$. We use the variable SUM to store the result. The variable I is initialized to 1 and the variable SUM to 0. The algorithm works by first adding 1 into SUM, followed by 2, and then by 3 and so on. The process continues till N is added into SUM. The values 1, 2, 3, …, N are generated by using the variable I. If I is initially 1, it is then incremented to become 2, followed by 3, …, finally N. The following are the sequence of steps that would be executed one after another:

```
SUM - SUM + 1
SUM - SUM + 2
SUM - SUM + 3
       .
       .
       .
SUM - SUM + N
```

Instead of writing out N addition statements (note that the value of N is not known at development time) one after another, we use the following form:

```
SUM - SUM + I, for all I - 1 to N
```

We monotonically increment the value of I and repeat the execution of the statement $SUM = SUM + I$ until the last value of I, viz. N, has been added into SUM. Incrementing I next would cause it to hold $N + 1$. This is when the condition at Step 4 'I <= N' fails and the control of the algorithm moves to step 5. The execution of the statements 'SUM = SUM + I' and 'I = I + 1' would be repeated while 'I<=N'. The control of the algorithm remains within the loop (iteration) till the condition 'I <= N' evaluates to true. In

writing this algorithm, a strong assumption has been made, that is, the user always inputs a positive whole number for N.

Iterative constructs are thus used to simulate repetition. They are used when one or more statements need to be executed repeatedly. Consider the situation where we want to display the word 'HELLO' 100 times. We can write DISPLAY 'HELLO' 100 times as follows:

```
DISPLAY 'HELLO'   (1st. time)
DISPLAY 'HELLO'   (2nd. time)

DISPLAY 'HELLO'   (3rd. time)
        .
        .
        .
DISPLAY 'HELLO'   (100th. time)
```

Here we need to repeat just one statement 100 times. Similarly, if we want to repeat the execution of an entire statement block 10000 times or even more, imagine the length of such an algorithm. Writing codes in such a manner may become a bit cumbersome. Instead we can put the statement in an iterative construct and iterate (repeat) it 100 times. A variable such as I would be used to keep a track of the number of times the iteration has taken place. The modified construct would be as follows:

```
REPEAT steps (a)-(b) while I <- 100
        (a) DISPLAY 'HELLO'
        (b) I - I + 1
END WHILE
```

Like before, I would have to be initialized to 1.

Pseudocode

Pseudocodes are a means to represent an algorithm in a coded form. The specifications used in a pseudocode are syntactically not stringent as in a program code. However, they represent the steps of an algorithm in a coded form which very closely resembles program code. Given a pseudocode, translation of the same into program code is relatively easy. A pseudocode may be considered to be an intermediate code between an algorithm and program code. The pseudocode for the algorithm in Example 4 is as follows:

```
Pseudocode SUMN
{
        print 'Enter a value for the number N: '
        scan N
        I - 1
        SUM - 0
        while I <- N
        {
                SUM - SUM + I
                I - I +1
        }
        print 'The sum is: ', SUM
}
```

It should be noted that a pseudocode is slightly more cryptic compared to an algorithm and is closer in form to a C program code. The C code for the pseudocode shown here would be as follows:

```
#include <stdio.h>
main()
{
        int I, SUM, N;
        printf('\nEnter a value for the number N:');
        scanf('%d', &N);

                while(I <- N)
                {
                        SUM - SUM + I;
                        I - I + 1;
                }
                printf('\The sum is: %d', SUM);
}
```
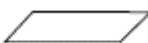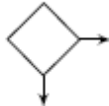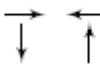
A quick comparison between the pseudocode and the corresponding program using the C language shows that function names such as 'printf' and 'scanf' have replaced pseudocodes such as 'print' and 'scan'. 'print' and 'scan' are codes for representing display and input operations. They are pseudo (false) because these codes would not be understood by the C compiler or the machine. Nevertheless, they are still in an encoded form. It is worth noting that the statements in the C program are syntactically bound. However, such restrictions are not to be found in a pseudocode.

Flowchart

Just like the algorithm, the flowchart is also a tool to represent logic. As the name indicates, a flowchart helps us graphically visualize the flow of control within the sequence of statements. At times it becomes easier for us to understand logic better when we have a picture in front of us. A flowchart proves to be really handy at such times. Unlike the algorithm, the flowchart uses some standard notation to graphically represent logic. Table 1.1 shows the graphical representation of some common constructs.

Table 1.1 Notations used in a flowchart

Name	Description	Symbol
Terminal symbol	This symbol is used to mark the beginning and end of a flowchart.	
Input/output box	This box is used to represent input operations or output operations.	
Process box	This box is used to represent processing operations. More specifically, this symbol is used to represent the imperative logic construct.	
Decision box	This diamond-shaped symbol is used to represent conditional statement constructs. Each such box has two exit points which correspond to the evaluation of the condition to true and false.	
Connector (intra-page)	This symbol is used to transfer the flow of control from one point to another within a page. A connector is usually labeled to match with its counterpart.	
Off-page connector (inter-page)	This symbol is used to transfer the flow of control from one point on a page to another point on a different page. This notation is useful when the flowchart spans more than one page.	
Arrowed lines	These are used to show the sequence of flow from one box to another.	

The symbols shown here are some basic ones that are used in flowcharts. Apart from these, there also exist other symbols to represent more constructs such as sub-routine call, reading from database and printing.

Let us now understand the use of these symbols to draw flowcharts. For ease of understanding, we consider the same set of examples which we have considered for understanding algorithms.

Flowchart AVERAGE
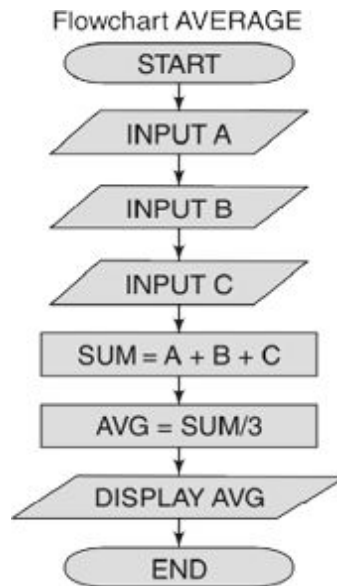
EXAMPLE 5 (Use of imperative constructs)

Problem – Develop an algorithm to find the average of three numbers taken as input from the user.

Algorithm AVERAGE

```
Step 0   START
Step 1   INPUT first number into variable A
Step 2   INPUT second number into variable B
Step 3   INPUT third number into variable C
Step 4   COMPUTE SUM - A + B + C
Step 5   COMPUTE AVG - SUM/3
Step 6   DISPLAY AVG
Step 7   END
```

The steps in the algorithm shown in the left match with the steps of the flowchart shown in the right.

Flowchart DIVISION
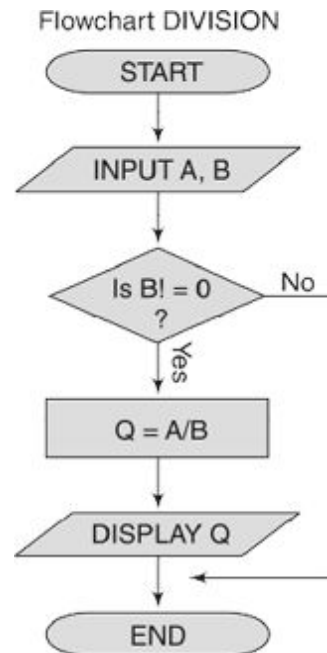
EXAMPLE 6 (Conditional construct)

Problem – Develop an algorithm to divide one number by another and find the quotient.

```
Algorithm DIVISION
Step 0  START
Step 1  INPUT first number into variable A
Step 2  INPUT second number into variable B
Step 3  IF B !- 0 THEN
                Q - A / B
                DISPLAY Q
        END IF
Step 4  END
```

In the flowchart shown on the right, notice the merging of two input operations into one input box. Also notice the use of the decision box. The calculation of the quotient and the display of the same takes place along the 'Yes' path, i.e. when the condition 'B != 0' evaluates to true. In case the condition evaluates to false, the control directly moves to the terminal box labeled 'END'.

Flowchart MAXIMUM
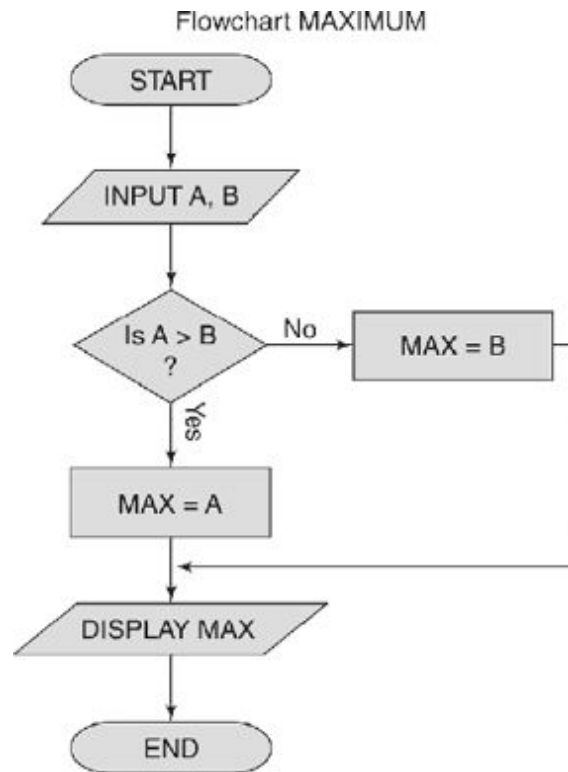
EXAMPLE 7 (Conditional construct)

Problem – Develop an algorithm to find the maximum of two numbers input by the user.

Algorithm MAXIMUM

```
Step 0  START
Step 1  INPUT first number into variable A
Step 2  INPUT second number into variable B
Step 3  IF A > B THEN
                MAX - A
        ELSE
                MAX - B
        END IF
Step 4  DISPLAY MAX
Step 5  END
```

The flowchart shown on the right computes the maximum of two numbers A and B. Do note the separate actions that take place along the Yes and the No paths. Also note how the display of the MAX takes place irrespective of whether A or B is the maximum. The Yes and the No paths merge before the display of the MAX, and then finally the flowchart terminates.

Flowchart SUM-N
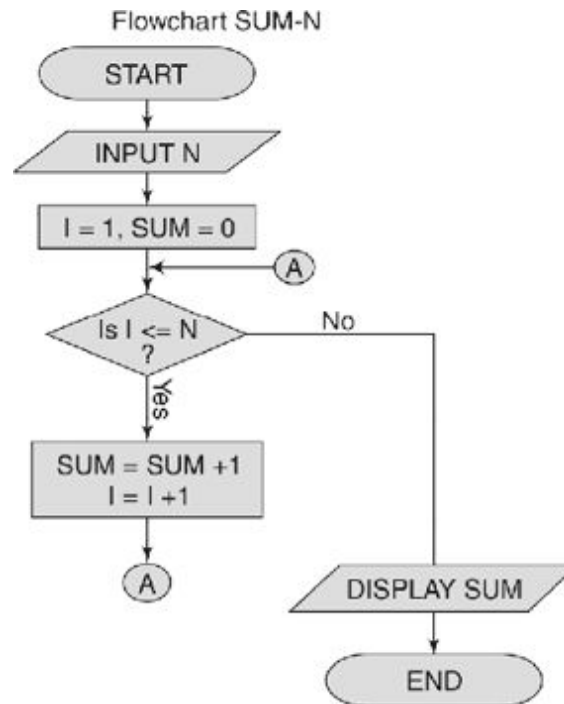
EXAMPLE 8 (Iteration construct)

Problem – Develop an algorithm to find the sum of the first N natural numbers where N would be input by the user.

```
Algorithm SUM-N

Step 0  START
Step 1  INPUT N
Step 2  I - 1
Step 3  SUM - 0
Step 4  REPEAT Steps (a) – (b) WHILE I <-N
                (a) SUM - SUM + I
                (b) I - I + 1
        END WHILE
Step 5  DISPLAY SUM
Step 6  END
```

Note the use of a connector labeled, 'A' in this flowchart.

Let us now consider an example which combines both an iterative as well as a conditional statement construct. For this example we will first write an algorithm, then draw an algorithm and finally transform the logic into a C program.

EXAMPLE 9

Problem – Write an algorithm, draw a flowchart, and finally write a C program for finding the sum of only the even numbers out of 'N' numbers

input by the user.

Algorithm SUM-EVEN

```
Step 0  START
Step 1  INPUT N
Step 2  I - 1
Step 3  SUM - 0
Step 4  REPEAT steps (a)-(d) WHILE I <- N
                (a) INPUT NUM
                (b) REM - NUM mod 2
                (c) IF REM -- 0
                        SUM - SUM + NUM
                    END IF
                        (d) I - I + 1
        END WHILE
Step 5  DISPLAY SUM
Step 6  END
```

Flow SUM-EVEN
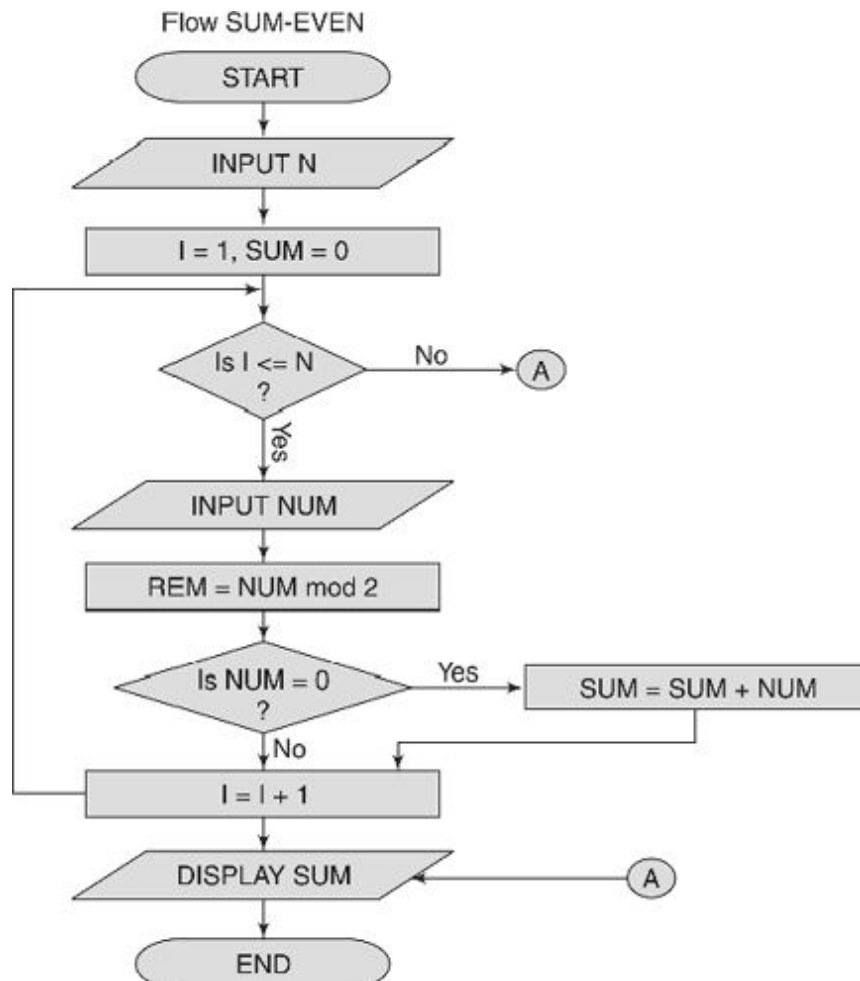
Let us now attempt to write the C code corresponding to the above algorithm/flowchart. Readers may skip this transformation part without any loss of continuity if they have difficulty understanding C syntactic

specifications. The transformation into a C program can be read after Chapter 2.

```
Program
  /* Program SUM-EVEN */
  #include <stdio.h>
  main() {
        int n, i, sum, num, rem;

        printf('\nEnter the number of elements : ');
        scanf('%d', &n);                          /*Step 1*/
        i - 1;                                    /*Step 2*/
        sum - 0;                                  /* Step 3*/

        while(i <-n) {                            /*Step 4 while loop begins*/
                printf('\nEnter data number %d : ', i);
                scanf('%d', &num);                /*(a) of loop*/
                rem - num % 2;                    /*(b) of loop (calculation of
                                                          remainder when num is
                                                          divided by 2)*/

                if (rem -- 0)                     /*(c) of loop (num is even)*/
                        sum - sum + num;          /*code if condition is true*/
                i - i + 1;                        /*(d) of loop*/
        }                                         /*Step 4 while loop ends
        printf('\nThe sum is %d', sum);           /*Step 5*/
  } /*end of main*/
```

On executing the above code after compilation and linking, the following will be the output for n = 3.

Output

Enter the number of elements : 3

Enter data number 1 : 40

Enter data number 2 : 57

Enter data number 3 : 78

The sum is 118

While writing the algorithm and the program it has been assumed that the count of the number of elements input by the user is greater or equal to 1. The logic could have been made more robust by checking whether the number input by the user in step 1 is greater or equal to 1 or not. An appropriate prompt may be displayed in case an invalid number has been input.

KEY CONCEPTS

- **HARDWARE:** The major Hardware components of a computer are the CPU, primary memory and input and output Devices. **[LO 1.1]**

- **SOFTWARE:** Two types of softwares are system and application software. **[LO 1.1]**

- **ALGORITHM:** An algorithm uses English-like statements to show the flow of logic in a step-wise manner. **[LO 1.2]**

- **FLOWCHART:** Flowchart is a graphical tool to represent logic flow. **[LO 1.2]**

ALWAYS REMEMBER

- The **central processing unit** is a VLSI circuit. **[LO 1.1]**

- The **control unit** generates control signals and sends them to the different components of a computer. **[LO 1.1]**

- The **arithmetic logic unit** has many circuits inside it that are capable of performing various arithmetic and logic operations. **[LO 1.1]**

- **Input devices** are used by the user to feed data and other information to the computer. **[LO 1.1]**

- **Output devices** are used by the computer to display an output or give the result of an operation to the user. **[LO 1.1]**

- **Memory** is of two types – the **primary** and the **secondary** memory. **[LO 1.1]**

- The **CPU** has direct connections to the **primary** memory. **[LO 1.1]**

- The **RAM** is a volatile memory. **[LO 1.1]**

- The **ROM** is a non-volatile memory. **[LO 1.1]**

- The secondary memory serves as a data reservoir. **[LO 1.1]**

- **System softwares** are those which interact with the hardware of the computer. **[LO 1.1]**

- The **operating system** acts as an interface between the user and the hardware. **[LO 1.1]**

- It is not possible to execute a program written in a high level language or assembly language directly without first translating it.
- A **high level language** uses a **compiler** for translation. **[LO 1.1]**
- A compiler is always typical of a programming language. **[LO 1.1]**
- Before writing a program, the **algorithm** and/or the **[LO 1.1]** **flowchart** for the same should be developed. **[LO 1.2]**
- Algorithms and flowcharts should be independent of programming languages. **[LO 1.2]**
- The three main programming constructs are **imperative, conditional and iterative** statements. **[LO 1.2]**
- Algorithms do not have a fixed style of writing. **[LO 1.2]**
- Algorithms should be finite and correct and the statements in it should be unambiguous. **[LO 1.2]**
- Flowcharts give us a visual depiction of logic flow. **[LO 1.2]**

REVIEW QUESTIONS

1.1 Write an algorithm and draw a flowchart to find the area of a circle of a given radius. **[LO 1.2 E]**

1.2 Write an algorithm and draw a flowchart to find the roots of a quadratic equation. **[LO 1.2 M]**

1.3 Write an algorithm and draw a flowchart to compute the simple interest on principal P, rate R% and duration T years. **[LO 1.2 E]**

1.4 Write an algorithm and draw a flowchart to compute the amount accrued if a sum of P is invested at a rate of R% for a duration of T years, where the interest is compounded every year. **[LO 1.2 E]**

1.5 Write an algorithm and draw a flowchart to find the sum of the digits of a 3-digit number. **[LO 1.2 M]**

1.6 Write an algorithm and draw a flowchart to find out whether a year is a leap year. **[LO 1.2 E]**

1.7 Write an algorithm and draw a flowchart to assign the grades of a student as per the following rules: **[LO 1.2 M]**

Marks	Grade
>= 90	O
>= 80 and < 90	E
>= 70 and < 80	A
>= 60 and < 70	B
>= 50 and < 60	C
>= 40 and < 50	D
< 40	F

1.8 Write an algorithm and draw a flowchart to find out whether three points (x1, y1), (x2, y2) and (x3, y3) lie on a straight line. **[LO 1.2 M]**

1.9 Write an algorithm and draw a flowchart to find out **[LO 1.2 M]** the age of a person correct to number of day(s), month(s) and year(s). **[LO 1.2 M]**

1.10 Write an algorithm and draw a flowchart to count the number of digits present in a number. **[LO 1.2 M]**

1.11 Write an algorithm and draw a flowchart to find the sum of the following series:

$$2 + 4 + 6 + \ldots + N \text{ terms}$$ **[LO 1.2 E]**

1.12 Write an algorithm and draw a flowchart to find the sum of the following series: $1! + 2 + 3! + \ldots + N!$ **[LO 1.2 M]**

1.13 Write an algorithm and draw a flowchart to find the sum of the following series: $1 \times 2 + 2 \times 3 + 3 \times 4 + \ldots + N \text{ terms}$ **[LO 1.2 M]**

1.14 Write an algorithm and draw a flowchart to find out whether a number is an Armstrong number. **[LO 1.2 E]**

1.15 Write an algorithm and draw a flowchart to find the sum of all prime numbers between 1 and 200. **[LO 1.2 H]**

1.16 Write an algorithm and draw a flowchart to display in words the digits of a number. **[LO 1.2 M]**

1.17 Write an algorithm and draw a flowchart to find out the sum of the following series:

$$1^2 + 3^2 + 5^2 + \ldots + N \text{ terms}$$ **[LO 1.2 E]**

E for Easy, **M** for Medium and **H** for High

Overview of C

Upon completing this chapter, you will be able to:

LO 2.1   Produce an overview of C programming language

LO 2.2   Exemplify the elementary C concepts through sample programs

LO 2.3   Illustrate the use of user-defined functions and math functions through sample programs

LO 2.4   Describe the basic structure of C program

LO 2.5   Recognize the programming style of C language

LO 2.6   Describe how a C program is compiled and executed

HISTORY OF C

'C' seems a strange name for a programming language. But this strange sounding language is one of the most popular computer languages today because it is a structured, high-level, machine independent language. It allows software developers to develop programs without worrying about the hardware platforms where they will be implemented.

The root of all modern languages is ALGOL, introduced in the early 1960s. ALGOL was the first computer language to use a block structure. Although it never became popular in USA, it was widely used in Europe. ALGOL gave the concept of structured programming to the computer science community. Computer scientists like Corrado Bohm, Guiseppe Jacopini and Edsger Dijkstra popularized this concept during 1960s. Subsequently, several languages were announced.

In 1967, Martin Richards developed a language called BCPL (Basic Combined Programming Language) primarily for writing system software. In 1970, Ken Thompson created a language using many features of BCPL and called it simply B. B was used to create early versions of UNIX

operating system at Bell Laboratories. Both BCPL and B were "typeless" system programming languages.

C was evolved from ALGOL, BCPL and B by Dennis Ritchie at the Bell Laboratories in 1972. C uses many concepts from these languages and added the concept of data types and other powerful features. Since it was developed along with the UNIX operating system, it is strongly associated with UNIX. This operating system, which was also developed at Bell Laboratories, was coded almost entirely in C. UNIX is one of the most popular network operating systems in use today and the heart of the Internet data superhighway.

For many years, C was used mainly in academic environments, but eventually with the release of many C compilers for commercial use and the increasing popularity of UNIX, it began to gain widespread support among computer professionals. Today, C is running under a variety of operating system and hardware platforms.

During 1970s, C had evolved into what is now known as *"traditional C"*. The language became more popular after publication of the book *'The C Programming Language'* by Brian Kerningham and Dennis Ritchie in 1978. The book was so popular that the language came to be known as "K&R C" among the programming community. The rapid growth of C led to the development of different versions of the language that were similar but often incompatible. This posed a serious problem for system developers.

To assure that the C language remains standard, in 1983, American National Standards Institute (ANSI) appointed a technical committee to define a standard for C. The committee approved a version of C in December 1989 which is now known as ANSI C. It was then approved by the International Standards Organization (ISO) in 1990. This version of C is also referred to as C89.

During 1990's, C++, a language entirely based on C, underwent a number of improvements and changes and became an ANSI/ISO approved language in November 1977. C++ added several new features to C to make it not only a true object-oriented language but also a more versatile language. During

the same period, Sun Microsystems of USA created a new language **Java** modelled on C and C++.
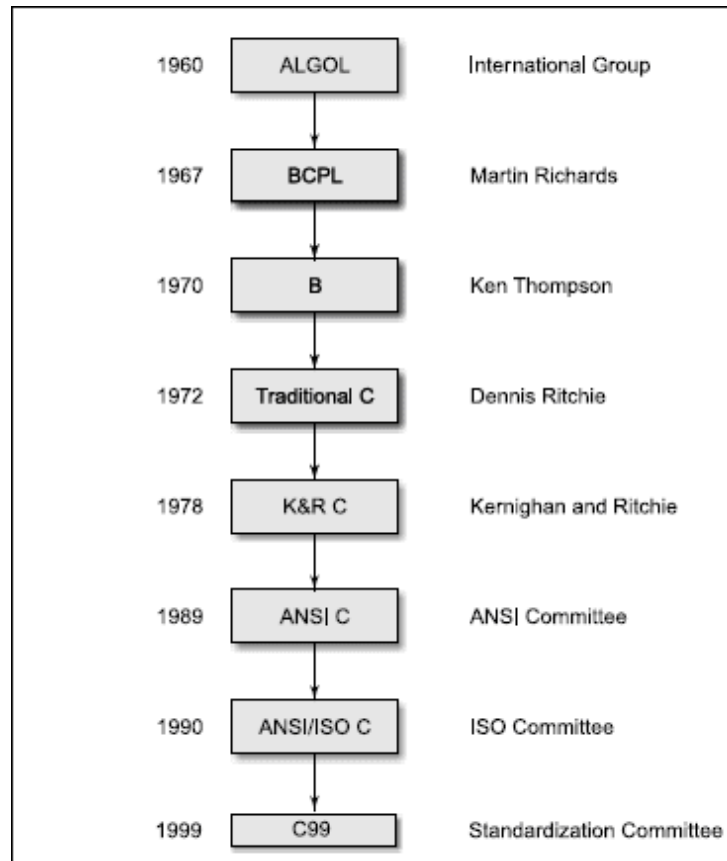
Fig. 2.1 History of ANSI C

All popular computer languages are dynamic in nature. They continue to improve their power and scope by incorporating new features and C is no exception. Although C++ and Java were evolved out of C, the standardization committee of C felt that a few features of C++/Java, if added to C, would enhance the usefulness of the language. The result was the 1999 standard for C. This version is usually referred to as C99. The history and development of C is illustrated in Fig. 2.1.

Although C99 is an improved version, still many commonly available compilers do not support all of the new features incorporated in C99. We, therefore, discuss all the new features added by C99 in an appendix separately so that the readers who are interested can quickly refer to the new material and use them wherever possible.

IMPORTANCE OF C

The increasing popularity of C is probably due to its many desirable qualities. It is a robust language whose rich set of built-in functions and operators can be used to write any complex program. The C compiler

combines the capabilities of an assembly language with the features of a high-level language and therefore it is well suited for writing both system software and business packages. In fact, many of the C compilers available in the market are written in C.

Programs written in C are efficient and fast. This is due to its variety of data types and powerful operators. It is many times faster than BASIC. For example, a program to increment a variable from 0 to 15000 takes about one second in C while it takes more than 50 seconds in an interpreter BASIC.

There are only 32 keywords in ANSI C and its strength lies in its built-in functions. Several standard functions are available which can be used for developing programs.

C is highly portable. This means that C programs written for one computer can be run on another with little or no modification. Portability is important if we plan to use a new computer with a different operating system.

C language is well suited for structured programming, thus requiring the user to think of a problem in terms of function modules or blocks. A proper collection of these modules would make a complete program. This modular structure makes program debugging, testing and maintenance easier.

Another important feature of C is its ability to extend itself. A C program is basically a collection of functions that are supported by the C library. We can continuously add our own functions to C library. With the availability of a large number of functions, the programming task becomes simple.

Before discussing specific features of C, we shall look at some sample C programs, and analyze and understand how they wok.

SAMPLE PROGRAM 1: PRINTING A MESSAGE

Consider a very simple program given in Fig. 2.2.

```
#include <stdio.h>

main( )
{
/*......... printing begins............*/
  printf('I see, I remember');
/*...........printing ends...............*/
}
```

Fig. 2.2 A program to print one line of text

This program when executed will produce the following output:

I see, I remember

Let us have a close look at the program. The first line informs the system that the name of the program is **main** and the execution begins at this line. The **main( )** is a special function used by the C system to tell the computer where the program starts. Every program must have *exactly one main* function. If we use more than one **main** function, the compiler cannot understand which one marks the beginning of the program.

The empty pair of parentheses immediately following **main** indicates that the function **main** has no *arguments* (or parameters). The concept of arguments will be discussed in detail later when we discuss functions (in Chapter 10).

The opening brace "**{** " in the second line marks the beginning of the function **main** and the closing brace "**}**" in the last line indicates the end of the function. In this case, the closing brace also marks the end of the program. All the statements between these two braces form the *function body*. The function body contains a set of instructions to perform the given task.

In this case, the function body contains three statements out of which only the **printf** line is an executable statement. The lines beginning with **/*** and ending with ***/** are known as *comment* lines. These are used in a program to enhance its readability and understanding. Comment lines are not executable

statements and therefore anything between **/*** and ***/** is ignored by the compiler. In general, a comment can be inserted wherever blank spaces can occur—at the beginning, middle or end of a line—"but never in the middle of a word".

Although comments can appear anywhere, they cannot be nested in C. That means, we cannot have comments inside comments. Once the compiler finds an opening token, it ignores everything until it finds a closing token. The comment line

/* = = = =/* = = = = */ = = = = */

is not valid and therefore results in an error.

Since comments do not affect the execution speed and the size of a compiled program, we should use them liberally in our programs. They help the programmers and other users in understanding the various functions and operations of a program and serve as an aid to debugging and testing. We shall see the use of comment lines more in the examples that follow.

Let us now look at the **printf( )** function, the only executable statement of the program.

printf("I see, I remember");

printf is a predefined standard C function for printing output. *Predefined* means that it is a function that has already been written and compiled, and linked together with our program at the time of linking. The concepts of compilation and linking are explained later in this chapter. The **printf** function causes everything between the starting and the ending quotation marks to be printed out. In this case, the output will be:

I see, I remember

Note that the print line ends with a semicolon. *Every statement in C should end with a semicolon (;) mark.*

Suppose we want to print the above quotation in two lines as

I see,

I remember!

This can be achieved by adding another **printf** function as shown below:

printf("I see, \n");

printf("I remember !");

The information contained between the parentheses is called the *argument* of the function. This argument of the first **printf** function is "I see, \n" and the second is "I remember !". These arguments are simply strings of characters to be printed out.

Notice that the argument of the first **printf** contains a combination of two characters \ and **n** at the end of the string. This combination is collectively called the *newline* character. A newline character instructs the computer to go to the next (new) line. It is similar in concept to the carriage return key on a typewriter. After printing the character comma (,) the presence of the newline character **\n** causes the string "I remember !" to be printed on the next line. No space is allowed between \ and n.

If we omit the newline character from the first **printf** statement, then the output will again be a single line as shown below.

I see, I remember !

This is similar to the output of the program in <u>Fig. 2.2</u>. However, note that there is no space between, and I.

It is also possible to produce two or more lines of output by one **printf** statement with the use of newline character at appropriate places. For example, the statement

printf("I see,\n I remember !");

will output

I see,

I remember !

while the statement

printf("I\n.. see,\n… … … I\n… … … remember !");

will print out

I

.. see,

… … … I

… … … remember !

Note   *Some authors recommend the inclusion of the statement.*

#include <stdio.h>

at the beginning of all programs that use any input/output library functions. However, this is not necessary for the functions *printf* and *scanf* which have been defined as a part of the C language. See Chapter 5 for more on input and output functions.
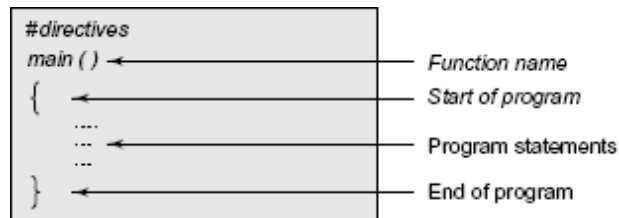
Fig. 2.3 Format of simple C programs

Before we proceed to discuss further examples, we must note one important point. C does make a distinction between *uppercase* and *lowercase* letters. For example, **printf** and **PRINTF** are not the same. In C, everything is written in lowercase letters. However, uppercase letters are used for symbolic names representing constants. We may also use uppercase letters in output strings like "I SEE" and "I REMEMBER".

The above example that printed **I see, I remember** is one of the simplest programs. Figure 2.3 highlights the general format of such simple programs. All C programs need a **main** function.

The *main* Function

The main is a part of every C program. C permits different forms of main statement. Following forms are allowed.

- main()
- int main()
- void main()
- main(void)
- void main(void)
- int main(void)

The empty pair of parentheses indicates that the function has no arguments. This may be explicitly indicated by using the keyword **void** inside the parentheses. We may also specify the keyword **int** or **void** before the word **main**. The keyword **void** means that the function does not return any information to the operating system and **int** means that the function returns an integer value to the operating system. When **int** is specified, the last statement in the program must be "return 0". For the sake of simplicity, we use the first form in our programs.

SAMPLE PROGRAM 2: ADDING TWO NUMBERS

Consider another program, which performs addition on two numbers and displays the result. The complete program is shown in Fig. 2.4.

```
/* Programm ADDITION                          line-1 */
/* Written by EBG                             line-2 */
main()                                /*      line-3 */
{                                     /*      line-4 */
int number;                           /*      line-5 */
float amount;                         /*      line-6 */
                                      /*      line-7 */
number = 100;                         /*      line-8 */
                                      /*      line-9 */
amount = 30.75 + 75.35;               /*      line-10 */
printf("%d\n",number);                /*      line-11 */
printf("%5.2f",amount);               /*      line-12 */
}                                     /*      line13 */
```

Fig. 2.4 Program to add two numbers

This program when executed will produce the following output:

100

106.10

The first two lines of the program are comment lines. It is a good practice to use comment lines in the beginning to give information such as name of the program, author, date, etc. Comment characters are also used in other lines to indicate line numbers.

The words **number** and **amount** are *variable names* that are used to store numeric data. The numeric data may be either in *integer* form or in *real* form. In C, *all variables should be declared* to tell the compiler what the **variable names** are and what **type of data** they hold. The variables must be declared before they are used. In lines 5 and 6, the declarations

```
int number;
float amount;
```

tell the compiler that **number** is an integer (**int**) and **amount** is a floating (**float**) point number. Declaration statements must appear at the beginning of the functions as shown in Fig. 2.4. All declaration statements end with a semicolon; C supports many other data types and they are discussed in detail in Chapter 3.

The words such as **int** and **float** are called the *keywords* and cannot be used as *variable* names. A list of keywords is given in Chapter 3.

Data is stored in a variable by *assigning* a data value to it. This is done in lines 8 and 10. In line-8, an integer value 100 is assigned to the integer variable **number** and in line-10, the result of addition of two real numbers 30.75 and 75.35 is assigned to the floating point variable **amount.** The statements

    number = 100;
    amount = 30.75 + 75.35;

are called the *assignment* statements. Every assignment statement must have a semicolon at the end.

The next statement is an output statement that prints the value of **number**. The print statement

    printf("%d\n", number);

contains two arguments. The first argument "%d" tells the compiler that the value of the second argument **number** should be printed as a *decimal integer*. Note that these arguments are separated by a comma. The newline character \n causes the next output to appear on a new line.

The last statement of the program

    printf("%5.2f", amount);

prints out the value of **amount** in floating point format. The format specification *%5.2f* tells the compiler that the output must be in *floating point,* with five places in all and two places to the right of the decimal point.

SAMPLE PROGRAM 3: INTEREST CALCULATION

The program in Fig. 2.5 calculates the value of money at the end of each year of investment, assuming an interest rate of 11 percent and prints the year, and the corresponding amount, in two columns. The output is shown in Fig. 2.6 for a period of 10 years with an initial investment of 5000.00. The program uses the following formula:

Value at the end of year = Value at start of year (1 + interest rate)

In the program, the variable **value** represents the value of money at the end of the year while **amount** represents the value of money at the start of the year. The statement

amount = value ;

makes the value at the end of the *current* year as the value at start of the *next* year.

```
/*————— INVESTMENT PROBLEM —————*/
#define PERIOD 10
#define PRINCIPAL   5000.00
/*————— MAIN PROGRAM BEGINS —————*/
main()
{ /*————— DECLARATION STATEMENTS —————*/
```

```
        int year;
        float amount, value, inrate;
/*————— ASSIGNMENT STATEMENTS —————*/
        amount - PRINCIPAL;
        inrate - 0.11;
        year - 0;
/*————— COMPUTATION STATEMENTS —————*/
/*————— COMPUTATION USING While LOOP —————*/
        while(year <- PERIOD)
        { printf("%2d  %8.2f\n",year, amount);
            value - amount + inrate * amount;
            year - year + 1;
            amount - value;
     }
/*————— while LOOP ENDS —————*/
}
/*————— PROGRAM ENDS —————*/
```

Fig. 2.5 Program for investment problem

Let us consider the new features introduced in this program. The second and third lines begin with **#define** directives. A **#define** directive defines value to a *symbolic constant* for use in the program. Whenever a symbolic name is encountered, the compiler substitutes the value associated with the name automatically. To change the value, we have to simply change the definition. In this example, we have defined two symbolic constants **PERIOD** and **PRINCIPAL** and assigned values 10 and 5000.00 respectively. These values remain constant throughout the execution of the program.

```
        0        5000.00
        1        5550.00
        2        6160.50
        3        6838.15
        4        7590.35
        5        8425.29
        6        9352.07
        7       10380.00
        8       11522.69
        9       12790.00
       10       14197.11
```

Fig. 2.6 Output of the investment program

The *#define* Directive

A **#define** is a preprocessor compiler directive and not a statement. Therefore **#define** lines should not end with a semicolon. Symbolic constants are generally written in uppercase so that they are easily distinguished from lowercase variable names. **#define** directives are usually placed at the beginning before the **main()** function. Symbolic constants are not declared in declaration section. Preprocessor directives are discussed in Chapter 15.

We must note that the defined constants are not variables. We may not change their values within the program by using an assignment statement. For example, the statement

<div align="center">

PRINCIPAL = 10000.00;

</div>

is illegal.

The declaration section declares **year** as integer and **amount, value** and **inrate** as floating point numbers. Note all the floating-point variables are declared in one statement. They can also be declared as

> **float amount;**
>
> **float value;**
>
> **float inrate;**

When two or more variables are declared in one statement, they are separated by a comma.

All computations and printing are accomplished in a **while** loop. **while** is a mechanism for evaluating repeatedly a statement or a group of statements. In this case as long as the value of **year** is less than or equal to the value of **PERIOD**, the four statements that follow **while** are executed. Note that these four statements are grouped by braces. We exit the loop when **year** becomes greater than **PERIOD**. The concept and types of loops are discussed in Chapter 7.

C supports the basic four arithmetic operators (−, +, *, /) along with several others. They are discussed in Chapter 4.

SAMPLE PROGRAM 4: USE OF SUBROUTINES

So far, we have used only **printf** function that has been provided for us by the C system. The program shown in Fig. 2.7 uses a user-defined function. A function defined by the user is equivalent to a subroutine in FORTRAN or subprogram in BASIC.

> **LO 2.3**
> Illustrate the use of user-defined functions and math functions through sample programs

Figure 2.7 presents a very simple program that uses a **mul ( )** function. The program will print the following output.

Multiplication of 5 and 10 is 50

```
/*————— PROGRAM USING FUNCTION ————*/
int mul (int a, int b); /*—— DECLARATION ——*/
/*————— MAIN PROGRAM BEGINS ————*/
      main ()
      {
            int a, b, c;
            a - 5;
            b - 10;
            c - mul (a,b);
```

```
            printf ("multiplication of %d and %d is %d",a,b,c);
      }
/* ————             MAIN PROGRAM ENDS
                    MUL() FUNCTION STARTS ———————*/
      int mul (int x, int y)
      int p;
      {
            p - x*y;
            return(p);
      }
/* ————————— MUL () FUNCTION ENDS ————*/
```

Fig. 2.7 A program using a user-defined function

The **mul ( )** function multiplies the values of **x** and **y** and the result is returned to the **main ( )** function when it is called in the statement

c = mul (a, b);

The **mul ( )** has two *arguments* **x** and **y** that are declared as integers. The values of **a** and **b** are passed on to **x** and **y** respectively when the function **mul ( )** is called. User-defined functions are considered in detail in Chapter 10.

SAMPLE PROGRAM 5: USE OF MATH FUNCTIONS

We often use standard mathematical functions such as cos, sin, exp, etc. We shall see now the use of a mathematical function in a program. The standard mathematical functions are defined and kept as a part of C **math library**. If we want to use any of these mathematical functions, we must add an **#include** instruction in the program. Like **#define,** it is also a compiler directive that instructs the compiler to link the specified mathematical functions from the library. The instruction is of the form

#include <math.h>

math.h is the filename containing the required function. illustrates the use of cosine function. The program calculates cosine values for angles 0, 10, 20………….180 and prints out the results with headings.

```
/*———— PROGRAM USING COSINE FUNCTION ———— */
#include <math.h>
#define    PI 3.1416
#define    MAX   180
main ( )
{
       int angle;
       float x,y;
       angle - 0;
       printf("   Angle     Cos(angle)\n\n");
       while(angle <- MAX)
```

```
                {
                     x - (PI/MAX)*angle;
                     y - cos(x);
                     printf("%15d %13.4f\n", angle, y);
                     angle - angle + 10;
                }
           }
Output
                     Angle              Cos(angle)
                       0                  1.0000
                      10                  0.9848
                      20                  0.9397
                      30                  0.8660
                      40                  0.7660
                      50                  0.6428
                      60                  0.5000
                      70                  0.3420
                      80                  0.1736
                      90                 -0.0000
                     100                 -0.1737
                     110                 -0.3420
                     120                 -0.5000
                     130                 -0.6428
                     140                 -0.7660
                     150                 -0.8660
                     160                 -0.9397
                     170                 -0.9848
                     180                 -1.0000
```

Fig. 2.8 Program using a math function

Another **#include** instruction that is often required is

#include <stdio.h>

stdio.h refers to the *standard* I/O header file containing standard input and output functions

The *#include* Directive

As mentioned earlier, *C* programs are divided into modules or functions. Some functions are written by users, like us, and many others are stored in the *C* library. Library functions are grouped category-wise and stored in different files known as *header files*. If we want to access the functions

stored in the library, it is necessary to tell the compiler about the files to be accessed.

This is achieved by using the preprocessor directive **#include** as follows:

#include<filename>

filename is the name of the library file that contains the required function definition. Preprocessor directives are placed at the beginning of a program.

A list of library functions and header files containing them are given in Appendix III.

BASIC STRUCTURE OF C PROGRAMS

The examples discussed so far illustrate that a C program can be viewed as a group of building blocks called *functions*. A function is a subroutine that may include one or more *statements* designed to perform a

LO 2.4
Describe the basic structure of C program

specific task. To write a C program, we first create functions and then put them together. A C program may contain one or more sections as shown in Fig. 2.9.

```
Documentation Section
Link Section (#include section)
Definition Section
Global Declaration Section
main ( ) Function Section
{
        Declaration part
        Executable part
}
Subprogram section

        Function 1
        Function 2
          –                    (User-defined functions)
          –
        Function n
```
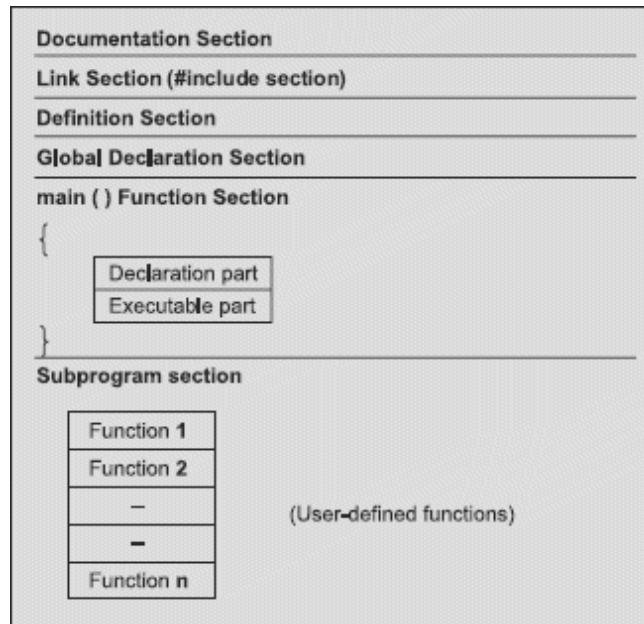
Fig. 2.9 An overview of a C program

The documentation section consists of a set of comment lines giving the name of the program, the author and other details, which the programmer would like to use later. The link section provides instructions to the compiler to link functions from the system library. The definition section defines all symbolic constants.

There are some variables that are used in more than one function. Such variables are called *global* variables and are declared in the *global* declaration section that is outside of all the functions. This section also declares all the user-defined functions.

Every C program must have one **main()** function section. This section contains two parts, declaration part and executable part. The declaration part declares all the variables used in the executable part. There is at least one statement in the executable part. These two parts must appear between the opening and the closing braces. The program execution begins at the opening brace and ends at the closing brace. The closing brace of the main function section is the logical end of the program. All statements in the declaration and executable parts end with a semicolon(;).

The subprogram section contains all the user-defined functions that are called in the **main** function. User-defined functions are generally placed immediately after the **main** function, although they may appear in any order.

All sections, except the **main** function section may be absent when they are not required.

PROGRAMMING STYLE

Unlike some other programming languages (COBOL, FORTRAN, etc.,) C is a *free-form* language. That is, the C compiler does not care, where on the line we begin typing. While this may be a licence for bad programming, we should try to use this fact to our advantage in developing readable programs. Although several alternative styles are possible, we should select one style and use it with total consistency.

LO 2.5
Recognize the
Programming style of C
language

First of all, we must develop the habit of writing programs in lowercase letters. C program statements are written in lowercase letters. Uppercase letters are used only for symbolic constants.

Braces, group program statements together and mark the beginning and the end of functions. A proper indentation of braces and statements would make a program easier to read and debug. Note how the braces are aligned and the statements are indented in the program of Fig. 2.5.

Since C is a free-form language, we can group statements together on one line. The statements

```
a = b;
x = y + 1;
z = a + x;
```

can be written on one line as

```
a = b; x = y+1; z = a+x;
```

The program

```
main( )
{
        printf("hello C");
}
```

may be written in one line like

```
main( ) {printf("Hello C")};
```

However, this style make the program more difficult to understand and should not be used. In this book, each statement is written on a separate line.

The generous use of comments inside a program cannot be overemphasized. Judiciously inserted comments not only increase the readability but also help to understand the program logic. This is very important for debugging and testing the program.

EXECUTING A 'C' PROGRAM

Executing a program written in C involves a series of steps. These are:

1. Creating the program;

2. Compiling the program;

3. Linking the program with functions that are needed from the C library; and

4. Executing the program.

Figure 2.10 illustrates the process of creating, compiling and executing a C program. Although these steps remain the same irrespective of the *operating system,* system commands for implementing the steps and conventions for naming *files* may differ on different systems.

An operating system is a program that controls the entire operation of a computer system. All input/output operations are channeled through the operating system. The operating system, which is an interface between the hardware and the user, handles the execution of user programs.
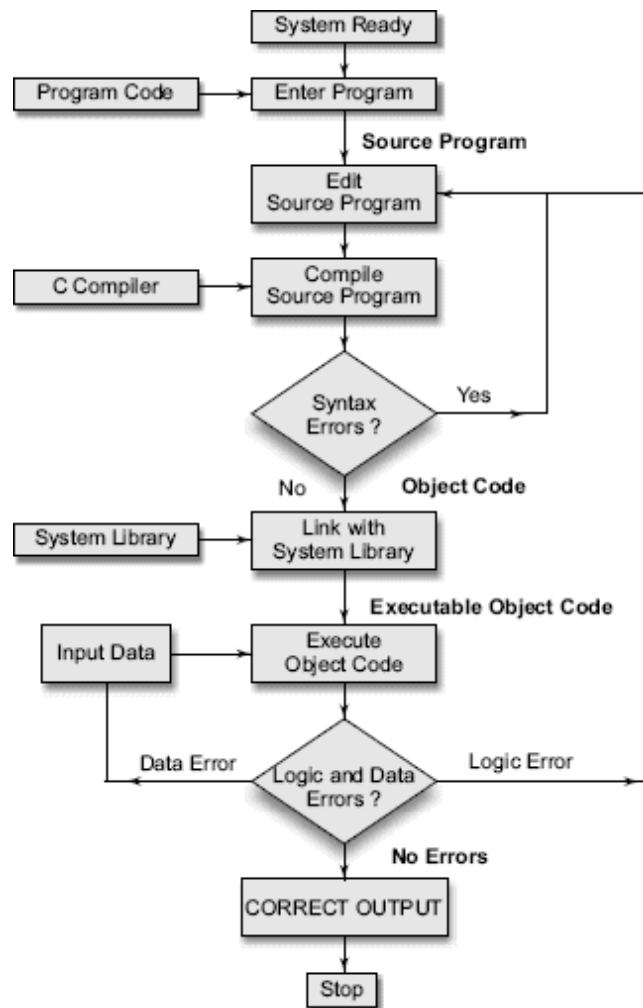
Fig. 2.10 Process of compiling and runnig a C program

Source code is the file containing the C source program. This program is input to a compiler. The compiler is a system software that translates the source program to **object code**. The object code contains an equivalent set of machine instructions corresponding to the source program which is understood by the computer. A given C program may make use of several library functions. Thus, the object code of the program would have to be linked to the object codes of the library functions. The linker is a system software that links together all the object codes to generate the final **executable code**. This executable code is submitted to the operating system for execution whenever required. The generation of the executable code is shown in Fig. 2.11.
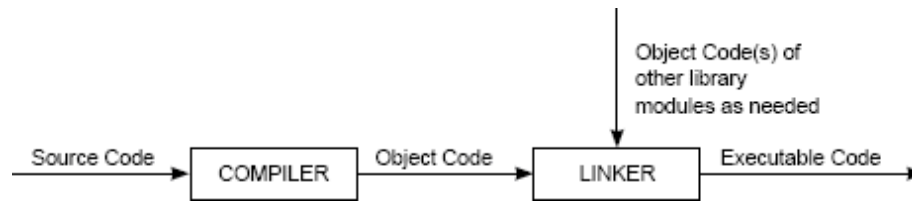
Fig. 2.11 Generation of executable code from source code

In this respect, it is worth noting that the compiler is specific to a programming language, and the object code is dependent upon the platform on which it is meant to run.

UNIX SYSTEM

Creating the Program

Once we load the UNIX operating system into the memory, the computer is ready to receive program. The program must be entered into a file. The file name can consist of letters, digits and special characters, followed by a dot and a letter **c**. Examples of valid file names are:

> LO 2.6
> Describe how a C program is compiled and executed

```
hello.c
program.c
ebg1.c
```

The file is created with the help of a *text editor*, either **ed** or **vi.** The command for calling the editor and creating the file is

<p align="center">ed <i>filename</i></p>

If the file existed before, it is loaded. If it does not yet exist, the file has to be created so that it is ready to receive the new program. Any corrections in the program are done under the editor. (The name of your system's editor may be different. Check your system manual.)

When the editing is over, the file is saved on disk. It can then be referenced any time later by its file name. The program that is entered into the file is known as the *source program*, since it represents the original form of the program.

Compiling and Linking

Let us assume that the source program has been created in a file named *ebg1.c.* Now the program is ready for compilation. The compilation command to achieve this task under UNIX is

cc *ebg1.c*

The source program instructions are now translated into a form that is suitable for execution by the computer. The translation is done after examining each instruction for its correctness. If everything is alright, the compilation proceeds silently and the translated program is stored on another file with the name *ebg1.o*. This program is known as *object code*.

Linking is the process of putting together other program files and functions that are required by the program. For example, if the program is using **exp()** function, then the object code of this function should be brought from the **math library** of the system and linked to the main program. Under UNIX, the linking is automatically done (if no errors are detected) when the **cc** command is used.

If any mistakes in the *syntax* and *semantics* of the language are discovered, they are listed out and the compilation process ends right there. The errors should be corrected in the source program with the help of the editor and the compilation is done again.

The compiled and linked program is called the *executable object code* and is stored automatically in another file named **a.out.**

Note that some systems use different compilation command for linking mathematical functions.

cc *filename* **- lm**

is the command under UNIPLUS SYSTEM V operating system.

Executing the Program

Execution is a simple task. The command

./a.out

would load the executable object code into the computer memory and execute the instructions. During execution, the program may request for some data to be entered through the keyboard. Sometimes the program does not produce the desired results. Perhaps, something is wrong with the program *logic* or *data*. Then it would be necessary to correct the source program or the data. In case the source program is modified, the entire process of compiling, linking and executing the program should be repeated.

Creating Your Own Executable File

Note that the linker always assigns the same name **a.out**. When we compile another program, this file will be overwritten by the executable object code of the new program. If we want to prevent from happening, we should rename the file immediately by using the command.

<div align="center">

mv a.out *name*

</div>

We may also achieve this by specifying an option in the cc command as follows:

<div align="center">

cc –o *name source-file*

</div>

This will store the executable object code in the file name and prevent the old file **a.out** from being destroyed.

Multiple Source Files

To compile and link multiple source program files, we must append all the files names to the **cc** command.

<div align="center">

cc *filename-***1.c** *…. filename-***n.c**

</div>

These files will be separately compiled into object files called

<div align="center">

filename-i.o

</div>

and then linked to produce an executable program file **a.out** as shown in Fig. 2.12.
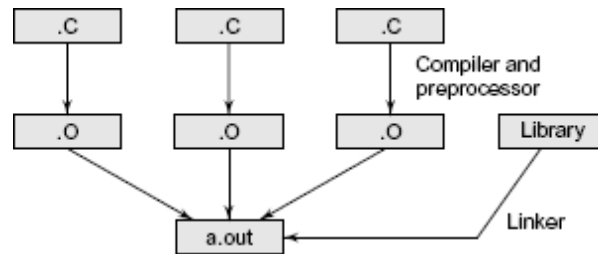
Fig. 2.12 Compilation of multiple files

It is also possible to compile each file separately and link them later. For example, the commands

cc –c mod1.c

cc –c mod2.c

will compile the source files *mod1.c* and *mod2.c* into objects files *mod1.o* and *mod2.o*. They can be linked together by the command

cc mod1.o mod2.o

we may also combine the source files and object files as follows:

cc mod1.c mod2.o

Only *mod1.c* is compiled and then linked with the object file mod2.o. This approach is useful when one of the multiple source files need to be changed and recompiled or an already existing object files is to be used along with the program to be compiled.

MS-DOS SYSTEM

The program can be created using any word processing software in non-document mode. The file name should end with the characters ".c" like **program.c, pay.c,** etc. Then the command

MSC pay.c

under MS-DOS operating system would load the program stored in the file **pay.c** and generate the **object code.** This code is stored in another file under name **pay.obj**. In case any language errors are found, the compilation is not completed. The program should then be corrected and compiled again.

The linking is done by the command

LINK pay.obj

which generates the **executable code** with the filename **pay.exe**. Now the command

<div align="center">**pay**</div>

would execute the program and give the results.

WINDOWS SYSTEM

Unlike in the Linux system, the Windows system does not come with an in-built C compiler. There are several IDEs. An IDE is an integrated development environment which provides programmers the ways and means to write, save, compile, edit, link and run programs. These IDEs have to be separately installed on a Windows system to be used. Typical examples are Borland C/C++, Microsoft Visual Studio, Dev-C++, Turbo C/C++, etc. Most of these IDEs have an extremely friendly graphic user interface. Thus, most of the work can be done by the click of a mouse. Some of these IDEs also work in the DOS environment. The IDE would then have to be invoked by opening the DOS box/command prompt. Once in the IDE environments, the tools offered by it would help us perform the required activities of compiling, linking and executing.

KEY CONCEPTS

- **#define:** Is a preprocessor compiler directive. **[LO 2.2]**

- **printf:** Is a predefined standard C function that writes the output to the stdout (standard output) stream. **[LO 2.2]**

- **scanf:** Is a predefined standard C function that reads formatted input from stdin (standard input) stream. **[LO 2.2]**

- **PROGRAM:** Is a sequence of instructions written to perform a specific task in the computer. **[LO 2.4]**

ALWAYS REMEMBER

- C is a structured, high-level, machine independent language. **[LO 2.1]**

- ANSI C and C99 are the standardized versions of C language. **[LO 2.1]**

- C combines the capabilities of assembly language with the features of a high level language. **[LO 2.1]**

- C is robust, portable and structured programming language. **[LO 2.1]**

- Every C program requires a **main()** function (Use of more than one **main()** is illegal). The place **main** is where the program execution begins. **[LO 2.2]**

- The execution of a function begins at the opening brace of the function and ends at the corresponding closing brace. **[LO 2.2]**

- C programs are written in lowercase letters. However, uppercase letters are used for symbolic names and output strings. **[LO 2.2]**

- All the words in a program line must be separated from each other by at least one space, or a tab, or a punctuation mark. **[LO 2.2]**

- Every program statement in a C language must end with a semicolon. **[LO 2.2]**

- All variables must be declared for their types before they are used in the program. **[LO 2.2]**

- A comment can be inserted almost anywhere a space can appear. Use of appropriate comments in proper places increases readability and understandability of the program and helps users in debugging and testing. Remember to match the symbols /* and * appropriately. **[LO 2.2]**

- Compiler directives such as **define** and **include** are special instructions to the compiler to help it compile a program. They do not end with a semicolon. **[LO 2.2]**

- The sign # of compiler directives must appear in the first column of the line. **[LO 2.2]**

- We must make sure to include header files using **#include** directive when the program refers to special names and functions that it does not define. **[LO 2.3]**

- The structure of a C program comprises of various sections including Documentation, Link, Definition, Global Declaration, main () function and Sub program section. **[LO 2.4]**

- C is a free-form language and therefore a proper form of indentation of various sections would improve legibility of the program. **[LO 2.5]**

- The execution of a C program involves a series of steps including: creating the program, compiling the program, linking the program with functions from C library and executing the program. **[LO 2.6]**

- The command used for running a C program in UNIX system is *a.out*. **[LO 2.6]**

- The command used for running a C program in MS-DOS system is *file.exe* where file is the name of the program that has already been compiled. **[LO 2.6]**

- When braces are used to group statements, make sure that the opening brace has a corresponding closing brace. **[LO 2.6]**

MULTIPLE CHOICE QUESTIONS

Choose the correct answer for each of the following multiple choice questions:

2.1 Which of the following characterize the typical features of C language? **[LO 2.1 M]**

(a) High level

(b) Machine independent

(c) Structured

(d) All of the above

2.2 A typical C program comprises the following sections:

- A - Documentation

- B - Definition

- C - Link

- D - main ()

- E - Subprogram section

- F - Global declaration

Identify the correct sequence of the C program sections:

2.3 Which of the following types of errors are detected **[LO 2.4 H]**
during program compilation? **[LO 2.6 M]**

  (a) Syntax

  (b) Logical

  (c) Data

  (d) All of the above

2.4 Consider the following C statements:

```
A    a-b;
     x -1;
     z-a+x;
B    a-b;x-1;z-a+x;
```

Which of the above programming styles are correct in C?

  (a) A is correct, B is incorrect **[LO 2.5 M]**

  (b) A is incorrect, B is correct

  (c) Both A and B are correct

  (d) Both A and B are incorrect

2.5 C language comprises _____ number of keywords. **[LO 2.1 E]**

  (a) 32

  (b) 16

  (c) 64

  (d) 128

REVIEW QUESTIONS

2.1 State whether the following statements are *true* or *false*.

  (a) Every line in a C program should end with a semicolon.

  (b) The closing brace of the **main( )** in a program is the **[LO 2.2 E]**
logical end of the program. **[LO 2.2 E]**

  (c) Comments cause the computer to print the text enclosed
between /* and */ when executed. **[LO 2.2 E]**

(d) Every C program ends with an END word.　　　**[LO 2.2 M]**

(e) A **printf** statement can generate only one line of output.
　　　　　　　　　　　　　　　　　　　　　　[LO 2.2 M]

(f) The purpose of the header file such as **stdio.h** is to store the source code of a program.　　**[LO 2.3 M]**

(g) A line in a program may have more than one statement.
　　　　　　　　　　　　　　　　　　　　　　[LO 2.5 M]

(h) Syntax errors will be detected by the compiler.　　**[LO 2.6 M]**

(i) In C language lowercase letters are significant.　**[LO 2.2 H]**

(j) **main( )** is where the program begins its execution. **[LO 2.2 H]**

(k) Every C program must have at least one user-defined function.　　　　　　　　　　　　　　　**[LO 2.3 E]**

(l) Declaration section contains instructions to the computer.
　　　　　　　　　　　　　　　　　　　　　　[LO 2.4 E]

(m) Only one function may be named **main( )**.**[LO 2.2 M]**

(n) Comments serve as internal documentation for programmers.
　　　　　　　　　　　　　　　　　　　　　　[LO 2.2 E]

(o) In C, we can have comments inside comments.　**[LO 2.2 M]**

(p) Use of comments reduces the speed of execution of a program.　　　　　　　　　　　　　　　　**[LO 2.2 H]**

(q) A comment can be inserted in the middle of a statement.
　　　　　　　　　　　　　　　　　　　　　　[LO 2.2 H]

2.2　Fill in the blanks with appropriate words in each of the following statements.

(a) Every program statement in a C program must end with a ＿＿＿＿＿＿＿＿.　　　　　　　　　　**[LO 2.2 E]**

(b) The ＿＿＿＿＿＿＿ Function is used to display the output on the screen.　　　　　　　　　　　　**[LO 2.2 E]**

(c) The ＿＿＿＿＿＿＿ header file contains mathematical functions.　　　　　　　　　　　　　　　**[LO 2.3 M]**

(d) The escape sequence character ＿＿＿＿＿＿ causes the cursor to move to the next line on the screen.　**[LO 2.2 H]**

(e) C programs are written in lower case letters whereas upper case letters are mainly used to define ＿＿＿＿＿**[LO 2.2 M]**

(f) C language offers several built-in ＿＿＿＿ that can be used in a program by including the relevant header files.**[LO 2.1 E]**

(g) ＿＿＿＿＿＿＿ indicates the starting point of a C program.

2.3 Remove the semicolon at the end of the **printf** statement **[LO 2.2 M]** in the program of [Fig. 2.2](#) and execute it. What is the output?

2.4 In the Sample Program 2, delete line-5 and execute the **[LO 2.2 M]** program. How helpful is the error message? **[LO 2.2 H]**

2.5 Modify the Sample Program 3 to display the following output: **[LO 2.2 M]**

```
Year              Amount
1                 5500.00
2                 6160.00
__               _____
__               _____
10                14197.11
```

2.6 Why and when do we use the **#define** directive? **[LO 2.2 E]**

2.7 Why and when do we use the **#include** directive? **[LO 2.3 M]**

2.8 What does **void main(void)** mean? **[LO 2.2 H]**

2.9 Distinguish between the following pairs:
   (a) main( ) and void main(void) **[LO 2.2 H]**
   (b) int main( ) and void main( ) **[LO 2.2 M]**

2.10 Why do we need to use comments in programs? **[LO 2.2 E]**

2.11 Why is the look of a program is important? **[LO 2.5 E]**

2.12 Where are blank spaces permitted in a C program? **[LO 2.5 sM]**

2.13 Describe the structure of a C program. **[LO 2.4 M]**

2.14 Describe the process of creating and executing a C program under UNIX system. **[LO 2.6 M]**

2.15 How do we implement multiple source program files? **[LO 2.6 H]**

DEBUGGING EXERCISES

2.1 Find errors, if any, in the following program: **[LO 2.2 E]**

```
/* A simple program
int main( )
{
    /* Does nothing */
}
```

2.2 Find errors, if any, in the following program: **[LO 2.2 M]**

```
#include (stdio.h)
void main(void)
{
    print('Hello C');
}
```

2.3   Find errors, if any, in the following program: **[LO 2.3 H]**

```
Include <math.h>
main { }
(
    FLOAT X;
    X - 2.5;
    Y - exp(x);
    Print(x,y);
)
```

PROGRAMMING EXERCISES

2.1   Write a program to display the equation of a line in the form

$$ax + by = c$$

for a = 5, b = 8 and c = 18.   **[LO 2.2 E]**

2.2   Write a C program that uses an in-built function to draw a 3D bar.
                                                                       [LO 2.3 E]

2.3   Write a program to output the following multiplication
      table: **[LO 2.2 M]**

```
5 × 1 = 5
5 × 2 = 10
5 × 3 = 15
   •        •
   •        •
5 × 10 = 50
```

2.4   Given the values of three variables a, b and c, write a program to
      compute and display the value of x, where

$$x = \frac{a}{b-c}$$

      Execute your program for the following values:

      (a) a = 250, b = 85, c = 25   **[LO 2.2 M]**
      (b) a = 300, b = 70, c = 70   **[LO 2.2 M]**

      Comment on the output in each case.

2.5   Write a C program that reads the value of distance travelled by a
      car and the time taken for the same. Next, compute the speed at

which the car travelled. **[LO 2.2 M]**

2.6 Write a C program to print the current system date. **[LO 2.3 M]**

2.7 Given two integers 20 and 10, write a program that uses a function add( ) to add these two numbers and sub( ) to find the difference of these two numbers and then display the sum and difference in the following form: **[LO 2.3 M]**

20 + 10 = 30

20 – 10 = 10

2.8 Modify the above program to provide border lines to the address. **[LO 2.2 H]**

2.9 Write a program using one print statement to print the **[LO 2.2 H]** pattern of asterisks as shown below: **[LO 2.2 H]**

```
*
*     *
*     *     *
*     *     *     *
```

2.10 Write a program that will print the following figure using suitable characters. **[LO 2.2 H]**
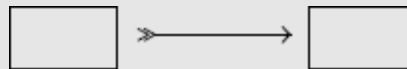
2.11 Area of a triangle is given by the formula

$$A = \sqrt{S(S-a)(S-b)(S-c)}$$

Where a, b and c are sides of the triangle and 2S = a + b + c. Write a program to compute the area of the triangle given the values of a, b and c. **[LO 2.2 H]**

2.12 Write a program to display the following simple arithmetic calculator **[LO 2.2 H]**

2.13 Distance between two points (x_1, y_1) and (x_2, y_2) is governed by the formula

$$D^2 = (x_2 - x_1)^2 + (y_2 - y_1)^2$$

Write a program to compute D given the coordinates of the points.

2.14 A point on the circumference of a circle whose center **[LO 2.3 H]** is (o, o) is (4,5). Write a program to compute perimeter and area of the circle. (Hint: use the formula given in the Ex. 2.11) **[LO 2.3 H]**

2.15 The line joining the points (2,2) and (5,6) which lie on the circumference of a circle is the diameter of the circle. Write a program to compute the area of the circle. **[LO 2.3 H]**

INTERVIEW QUESTIONS

2.1 Name some of the key features of C programming language.

2.2 Which symbol is used to indicate the end of most C **[LO 2.1 E]** programming statements? **[LO 2.2 E]**

2.3 Do you consider C as a high-level programming language?

2.4 Which punctuation symbols are used to represent code **[LO 2.1 M]** blocks in C? **[LO 2.2 M]**

2.5 Which function is the starting point in a C program? **[LO 2.2 M]**

2.6 Which return value of main() function indicates successful completion of the program? **[LO 2.2 H]**

E for Easy, **M** for Medium and **H** for High

Constants, Variables and Data Types

LEARNING OBJECTIVES

Upon completing this chapter, you will be able to:

LO 3.1   Know the C character set and keywords
LO 3.2   Describe constants and variables
LO 3.3   Identify the various C data types
LO 3.4   Discuss how variables are used in a program
LO 3.5   Explain how constants are used in a program

INTRODUCTION

A programming language is designed to help process certain kinds of *data* consisting of numbers, characters and strings and to provide useful output known as *information*. The task of processing of data is accomplished by executing a sequence of precise instructions called a *program*. These instructions are formed using certain symbols and words according to some rigid rules known as *syntax rules* (or *grammar*). Every program instruction must confirm precisely to the syntax rules of the language.

Like any other language, C has its own vocabulary and grammar. In this chapter, we will discuss the concepts of constants and variables and their types as they relate to C programming language.

CHARACTER SET

The characters that can be used to form words, numbers and expressions depend upon the computer on which the program is run. However, a subset of characters is available that can be used on most

LO 3.1
Know the C character set and keywords

personal, micro, mini and mainframe computers. The characters in C are grouped into the following categories:

1. Letters
2. Digits
3. Special characters
4. White spaces

The entire character set is given in Table 3.1.

The compiler ignores white spaces unless they are a part of a string constant. White spaces may be used to separate words, but are prohibited between the characters of keywords and identifiers.

Table 3.1 C Character Set

Letters		Digits
Uppercase A.....Z		All decimal digits 0 9
Lowercase a.....z		

Special Characters

, comma		& ampersand
. period		^ caret
; semicolon		* asterisk
: colon		– minus sign
? question mark		+ plus sign
' apostrophe		< opening angle bracket
" quotation mark		(or less than sign)
! exclamation mark		> closing angle bracket
	vertical bar		(or greater than sign)
/ slash		(left parenthesis
\ backslash		) right parenthesis
~ tilde		[left bracket
_ under score		] right bracket
$ dollar sign		{ left brace
% percent sign		} right brace
		# number sign

White Spaces

Blank space

Horizontal tab

Carriage return

New line

Form feed

Trigraph Characters

Table 3.2 ANSI C Trigraph Sequences

Trigraph sequence	Translation
??=	# number sign
??(	[left bracket
??)	] right bracket
??<	{ left brace
??>	} right brace
??!	| vetical bar
??/	\ back slash
??/	^ caret
??-	~ tilde

Many non-English keyboards do not support all the characters mentioned in Table 3.1. ANSI C introduces the concept of "trigraph" sequences to provide a way to enter certain characters that are not available on some keyboards. Each trigraph sequence consists of three characters (two question marks followed by another character) as shown in Table 3.2.

For example, if a keyboard does not support square brackets, we can still use them in a program using the trigraphs ??(and ??).

C TOKENS

In a passage of text, individual words and punctuation marks are called *tokens*. Similarly, in a C program the smallest individual units are known as C tokens. C has six types of tokens as shown in Fig. 3.1. C programs are written using these tokens and the syntax of the language.
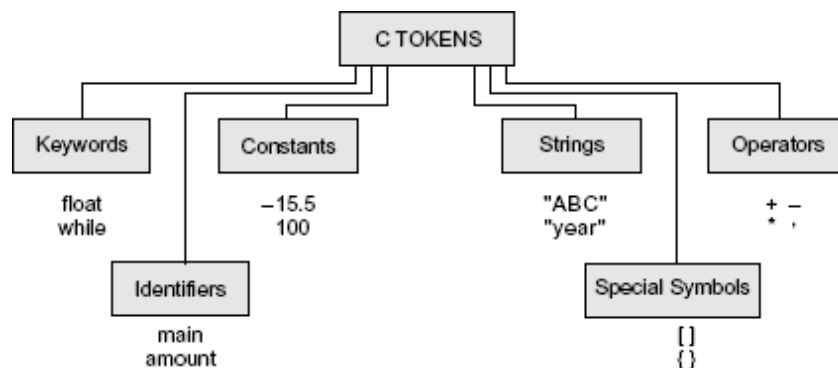
Fig. 3.1 C tokens and examples

KEYWORDS AND IDENTIFIERS

Every C word is classified as either a *keyword* or an *identifier*. All keywords have fixed meanings and these meanings cannot be changed. Keywords serve as basic building blocks for program statements. The list of all keywords of ANSI C are listed in Table 3.3. All keywords must be written in lowercase. Some compilers may use additional keywords that must be identified from the C manual.

Note *C99 adds some more keywords. See the Appendix "C99 Features".*

Identifiers refer to the names of variables, functions and arrays. These are user-defined names and consist of a sequence of letters and digits, with a letter as a first character. Both uppercase and lowercase letters are permitted, although lowercase letters are commonly used. The underscore character is also permitted in identifiers. It is usually used as a link between two words in long identifiers.

Table 3.3 ANSI C Keyword

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Rules for Identifiers

1. First character must be an alphabet (or underscore).
2. Must consist of only letters, digits or underscore.
3. Only first 31 characters are significant.
4. Cannot use a keyword.
5. Must not contain white space

CONSTANTS

Constants in C refer to fixed values that do not change during the execution of a program. C supports several types of constants as illustrated in Fig. 3.2.

Integer Constants

An *integer* constant refers to a sequence of digits. There are three types of integers, namely, *decimal* integer, *octal* integer and *hexadecimal* integer.

Decimal integers consist of a set of digits, 0 through 9, preceded by an optional – or + sign. Valid examples of decimal integer constants are:

123 – 321 0 654321 +78
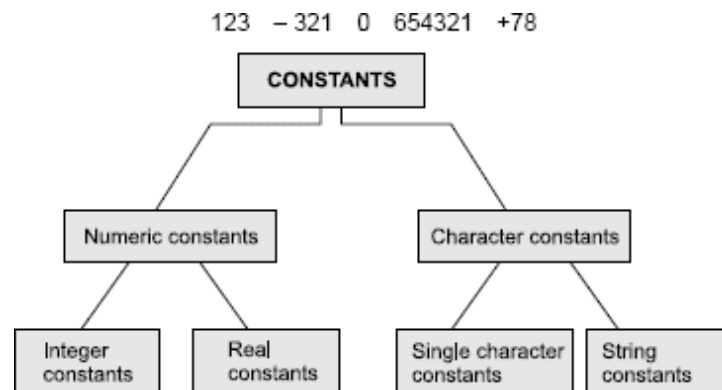
Fig. 3.2 Basic types of C constants

Embedded spaces, commas, and non-digit characters are not permitted between digits. For example,

15 750 20,000 $1000

are illegal numbers.

> **Note** ANSI C supports unary plus which was not defined earlier.

An *octal* integer constant consists of any combination of digits from the set 0 through 7, with a leading 0. Some examples of octal integer are:

037 0 0435 0551

A sequence of digits preceded by 0x or 0X is considered as *hexadecimal* integer. They may also include alphabets A through F or a through f. The letter A through F represent the numbers 10 through 15. Following are the examples of valid hex integers:

0X2 0x9F 0Xbcd 0x

We rarely use octal and hexadecimal numbers in programming.

The largest integer value that can be stored is machine-dependent. It is 32767 on 16-bit machines and 2,147,483,647 on 32-bit machines. It is also possible to store larger integer constants on these machines by appending *qualifiers* such as U,L and UL to the constants. Examples:

```
56789U          or 56789u          (unsigned integer)
987612347UL     or 98761234ul      (unsigned long integer)
9876543L        or 9876543l        (long integer)
```

The concept of unsigned and long integers are discussed in detail in Section 3.7.

WORKED-OUT PROBLEM 3.1 **E**

Representation of integer constants on a 16-bit computer.

The program in Fig. 3.3 illustrates the use of integer constants on a 16-bit machine. The output in Fig. 3.3 shows that the integer values larger than 32767 are not properly stored on a 16-bit machine. However, when they are qualified as long integer (by appending L), the values are correctly stored.

```
Program
        main()
        {
            printf("Integer values\n\n");
            printf("%d %d %d\n", 32767,32767+1,32767+10);
            printf("\n");
            printf("Long integer values\n\n");
            printf("%ld %ld %ld\n", 32767L,32767L+1L,32767L+10L);
        }
Output
        Integer values
        32767 -32768 -32759
        Long integer values
        32767  32768  3777
```

Fig. 3.3 Representation of integer constants on 16-bit machine

Real Constants

Integer numbers are inadequate to represent quantities that vary continuously, such as distances, heights, temperatures, prices, and so on. These quantities are represented by numbers containing fractional parts like 17.548. Such numbers are called *real* (or *floating point*) constants. Further examples of real constants are:

<div align="center">0.0083 –0.75 435.36 +247.0</div>

These numbers are shown in *decimal notation*, having a whole number followed by a decimal point and the fractional part. It is possible to omit digits before the decimal point, or digits after the decimal point. That is,

<div align="center">215. .95 –.71 +.5</div>

are all valid real numbers.

A real number may also be expressed in *exponential* (or *scientific) notation*. For example, the value 215.65 may be written as 2.1565e2 in exponential notation. e2 means multiply by 10^2. The general for is:

<div align="center">mantissa e exponent</div>

The *mantissa* is either a real number expressed in *decimal notation* or an integer. The *exponent* is an integer number with an optional *plus* or *minus sign*. The letter **e** separating the mantissa and the exponent can be written in either lowercase or uppercase. Since the exponent causes the decimal point to "float", this notation is said to represent a real number in *floating point form.* Examples of legal floating-point constants are:

<div align="center">0.65e4 12e-2 1.5e+5 3.18E3 -1.2E-1</div>

Embedded white space is not allowed.

Exponential notation is useful for representing numbers that are either very large or very small in magnitude. For example, 7500000000 may be written as 7.5E9 or 75E8. Similarly, -0.000000368 is equivalent to –3.68E-7.

Floating-point constants are normally represented as double-precision quantities. However, the suffixes f or F may be used to force single-precision and l or L to extend double precision further.

Some examples of valid and invalid numeric constants are given in Table 3.4.

Table 3.4 Examples of Numeric Constants

Constant	Valid?	Remarks
698354L	Yes	Represents long integer
25,000	No	Comma is not allowed
+5.0E3	Yes	(ANSI C supports unary plus)
3.5e-5	Yes	
7.1e 4	No	No white space is permitted
-4.5e-2	Yes	
1.5E+2.5	No	Exponent must be an integer
$255	No	$ symbol is not permitted
0X7B	Yes	Hexadecimal integer

Single Character Constants

A single character constant (or simply character constant) contains a single character enclosed within a pair of *single* quote marks. Example of character constants is as follows:

<p style="text-align:center">'5' 'X' ';' ' '</p>

Note that the character constant '5' is not the same as the *number 5*. The last constant is a blank space.

Character constants have integer values known as ASCII values. For example, the statement

would print the number 97, the ASCII value of the letter a. Similarly, the statement

```
printf("%c", 97);
```

would output the letter 'a'. ASCII values for all characters are given in Appendix II.

Since each character constant represents an integer value, it is also possible to perform arithmetic operations on character constants. They are discussed in Chapter 9.

String Constants

Table 3.5 Backslash Character Constants

Constant	Meaning
'\a'	audible alert (bell)
'\b'	back space
'\f'	form feed
'\n'	new line
'\r'	carriage return
'\t'	horizontal tab
'\v'	vertical tab
'\''	single quote
'\"'	double quote
'\?'	question mark
'\\'	backslash
'\0'	null

A string constant is a sequence of characters enclosed in *double* quotes. The characters may be letters, numbers, special characters and blank space. Examples are:

"Hello!" "1987" "WELL DONE" "?...!" "5+3" "X"

Remember that a character constant (e.g., 'X') is not equivalent to the single character string constant
(e.g., "X"). Further, a single character string constant does not have an equivalent integer value while a character constant has an integer value. Character strings are often used in programs to build meaningful programs. Manipulation of character strings are considered in detail in Chapter 9.

Backslash Character Constants

C supports some special backslash character constants that are used in output functions. For example, the symbol '\n' stands for newline character. A list of such backslash character constants is given in Table 3.5. Note that each one of them represents one character, although they consist of two characters. These characters combinations are known as *escape sequences*.

VARIABLES

A *variable* is a data name that may be used to store a data value. Unlike constants that remain unchanged during the execution of a program, a

variable may take different values at different times during execution. In Chapter 2, we used several variables. For instance, we used the variable **amount** in Worked Out Problem 3 to store the value of money at the end of each year (after adding the interest earned during that year).

A variable name can be chosen by the programmer in a meaningful way so as to reflect its function or nature in the program. Some examples of such names are:

Average

height

Total

Counter_1

class_strength

As mentioned earlier, variable names may consist of letters, digits, and the underscore(_) character, subject to the following conditions:

1. They must begin with a letter. Some systems permit underscore as the first character.

2. ANSI standard recognizes a length of 31 characters. (In C99, at least 63 characters are significant.)

3. Uppercase and lowercase are significant. That is, the variable **Total** is not the same as **total** or **TOTAL**.

4. It should not be a keyword.

5. White space is not allowed.

Some examples of valid variable names are:

John	Value	T_raise
Delhi	x1	ph_value
mark	sum1	distance

Invalid examples include:

123	(area)
%	25th

Further examples of variable names and their correctness are given in Table 3.6.

Table 3.6 Examples of Variable Names

Variable name	Valid ?	Remark
First_tag	Valid	
char	Not valid	char is a keyword
Price$	Not valid	Dollar sign is illegal
group one	Not valid	Blank space is not permitted
average_number	Valid	
int_type	Valid	Keyword may be part of a name

DATA TYPES

C language is rich in its *data types.* Storage representations and machine instructions to handle constants differ from machine to machine. The variety of data types available allow the programmer to select the type appropriate to the needs of the application as well as the machine.

> LO 3.3
> Identify the various C data types

ANSI C supports three classes of data types:

1. Primary (or fundamental) data types

2. Derived data types

3. User-defined data types

The primary data types and their extensions are discussed in this section. The user-defined data types are defined in the next section while the derived data types such as arrays, functions, structures and pointers are discussed as and when they are encountered.

All C compilers support five fundamental data types, namely integer **(int),** character (**char**), floating point **(float),** double-precision floating point **(double)** and **void.** Many of them also offer extended data types such as **long int** and **long double**. Various data types and the terminology used to describe them are given in Fig. 3.4. The range of the basic four types are given in Table 3.7. We discuss briefly each one of them in this section.

> **Note** C99 adds three more data types, namely _Bool, _Complex, and _Imaginary. See the Appendix "C99 Features".
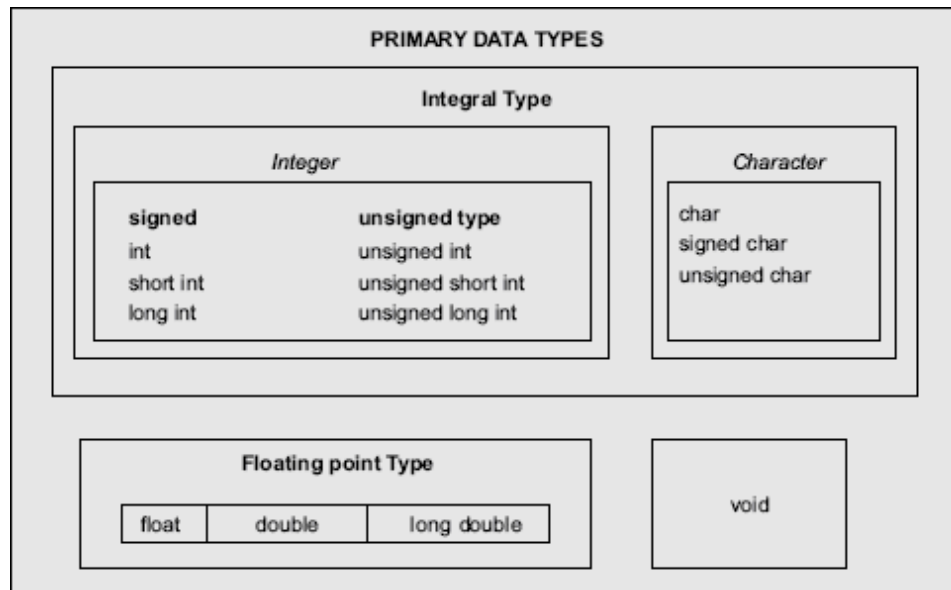
Fig. 3.4 Primary data types in C

Integer Types

Table 3.7 Size and Range of Basic Data Types on 16-bit Machines

Data type	Range of values
char	−128 to 127
int	−32,768 to 32,767
float	3.4e−38 to 3.4e+e38
double	1.7e−308 to 1.7e+308

Integers are whole numbers with a range of values supported by a particular machine. Generally, integers occupy one word of storage, and since the word sizes of machines vary (typically, 16 or 32 bits) the size of an integer that can be stored depends on the computer. If we use a 16 bit word length, the size of the integer value is limited to the range −32768 to +32767 (that is, −2^{15} to +2^{15}−1). A signed integer uses one bit for sign and 15 bits for the magnitude of the number. Similarly, a 32 bit word length can store an integer ranging from -2,147,483,648 to 2,147,483,647.
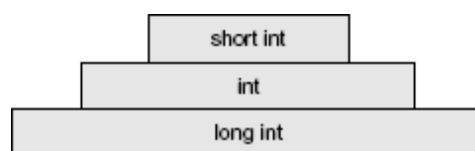
Fig. 3.5 Integer types

In order to provide some control over the range of numbers and storage space, C has three classes of integer storage, namely **short int, int,** and **long int,** in both **signed** and **unsigned** forms. ANSI C defines these types so that they can be organized from the smallest to the largest, as shown in . For example, **short int** represents fairly small integer values and requires half the amount of storage as a regular **int** number uses. Unlike signed integers, unsigned integers use all the bits for the magnitude of the number and are always positive. Therefore, for a 16 bit machine, the range of unsigned integer numbers will be from 0 to 65,535.

We declare **long** and **unsigned** integers to increase the range of values. The use of qualifier **signed** on integers is optional because the default declaration assumes a signed number. Table 3.8 shows all the allowed combinations of basic types and qualifiers and their size and range on a 16-bit machine.

Note C99 allows long long integer types. See the Appendix "C99 Features".

Table 3.8 Size and Range of Data Types on a 16-bit Machine

Type	Size (bits)	Range
char or signed char	8	−128 to 127
unsigned char	8	0 to 255
int or signed int	16	−32,768 to 32,767
unsigned int	16	0 to 65535
short int or		
signed short int	8	−128 to 127
unsigned short int	8	0 to 255
long int or		
signed long int	32	−2,147,483,648 to 2,147,483,647
unsigned long int	32	0 to 4,294,967,295
float	32	3.4E − 38 to 3.4E + 38
double	64	1.7E − 308 to 1.7E + 308
long double	80	3.4E − 4932 to 1.1E + 4932

Floating Point Types
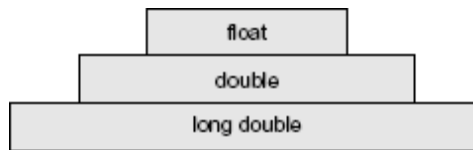
Fig. 3.6 Floating-point types

Floating point (or real) numbers are stored in 32 bits (on all 16 bit and 32 bit machines), with 6 digits of precision. Floating point numbers are defined in C by the keyword **float.** When the accuracy provided by a **float** number is not sufficient, the type **double** can be used to define the number. A **double** data type number uses 64 bits giving a precision of 14 digits. These are known as *double precision* numbers. Remember that double type represents the same data type that **float** represents, but with a greater precision. To extend the precision further, we may use **long double** which uses 80 bits. Floating point numbers are represented within the computer's memory using the IEEE standard. The relationship among floating types is illustrated Fig. 3.6.

Void Types

The **void** type has no values. This is usually used to specify the type of functions. The type of a function is said to be **void** when it does not return any value to the calling function. It can also play the role of a generic type, meaning that it can represent any of the other standard types.

Character Types

A single character can be defined as a **character(char)** type data. Characters are usually stored in 8 bits (one byte) of internal storage. The qualifier **signed** or **unsigned** may be explicitly applied to char. While **unsigned chars** have values between 0 and 255, **signed chars** have values from −128 to 127.

DECLARATION OF VARIABLES

After designing suitable variable names, we must declare them to the compiler. Declaration does two things:

LO 3.4
Discuss how variables are used in a program

1. It tells the compiler what the variable name is.

2. It specifies what type of data the variable will hold.

Declaration of variables must be done at the beginning of a statement block before they are used in a program.

Primary Type Declaration

A variable can be used to store a value of any data type. That is, the name has nothing to do with its type. The syntax for declaring a variable is as follows:

```
data-type v1,v2,....vn ;
```

v1, v2, ...vn are the names of variables. Variables are separated by commas. A declaration statement must end with a semicolon. For example, valid declarations are:

```
int count;
int number, total;
double ratio;
```

int and **double** are the keywords to represent integer type and real type data values respectively. Table 3.9 shows various data types and their keyword equivalents.

Table 3.9 Data Types and Their Keywords

Data type	Keyword equivalent
Character	char
Unsigned character	unsigned char
Signed character	signed char
Signed integer	signed int (or int)
Signed short integer	signed short int (or short int or short)
Signed long integer	signed long int (or long int or long)
Unsigned integer	unsigned int (or unsigned)
Unsigned short integer	unsigned short int (or unsigned short)
Unsigned long integer	unsigned long int (or unsigned long)
Floating point	float
Double-precision floating point	double
Extended double-precision floating point	long double

The program segment given in illustrates declaration of variables. **main()** is the beginning of the program. The opening brace { signals the execution of the program. Declaration of variables is usually done immediately after the opening brace of the program. The variables can also be declared outside (either before or after) the **main** function. The importance of place of declaration will be dealt in detail later while discussing functions.

```
main() /*.........Program Name........................ */
{
    /*...............Declaration.....................*/
        float           x, y;
        int             code;
        short int       count;
        long int        amount;
        double          deviation;
        unsigned        n;
        char            c;
    /*...............Computation..................... */
        . . . .
        . . . .
        . . . .
}  /*.............Program ends........................*/
```

Fig. 3.7 Declaration of variables

When an adjective (qualifier) **short, long, or unsigned** is used without a basic data type specifier, C compilers treat the data type as an **int.** If we want to declare a character variable as unsigned, then we must do so using both the terms like **unsigned char**.

Default Values of Constants

Integer constants, by default, represent **int** type data. We can override this default by specifying unsigned or long after the number (by appending U or L) as shown below:

Literal	Type	Value
+111	int	111
−222	int	−222
45678U	unsigned int	45,678
−56789L	long int	−56,789
987654UL	unsigned long int	9,87,654

Similarly, floating point constants, by default represent **double** type data. If we want the resulting data type to be **float** or **long double**, we must append the letter f or F to the number for **float** and letter l or L for **long double** as shown below:

Literal	Type	Value
0.	double	0.0
.0	double	0.0
12.0	double	12.0
1.234	double	1.234
−1.2f	float	−1.2
1.23456789L	long double	1.23456789

User-Defined Type Declaration

C supports a feature known as "type definition" that allows users to define an identifier that would represent an existing data type. The user-defined data type identifier can later be used to declare variables. It takes the general form:

```
typedef type identifier;
```

where *type* refers to an existing data type and "identifier" refers to the "new" name given to the data type. The existing data type may belong to any class of type, including the user-defined ones. Remember that the new type

is 'new' only in name, but not the data type. **typedef** cannot create a new type. Some examples of type definition are:

End of this sample Kindle book.
Enjoyed the sample?

BODY>