COMPUTER PROGRAMMING

A. K. Sharma

AGRIMOON.COM

COMPUTER PROGRAMMING

A. K. Sharma
Dairy Economics, Statistics and Management Division
NDRI, Karnal

Index

Lesson 1

PROBLEM SOLVING WITH COMPUTER PROGRAMMING – PART I (Algorithms and Flowcharts)

1.1 Introduction

A programming language is a systematic notation by which we describe computational processes to others. Computational process in present context means a set of steps that a machine can perform for solving a problem. However, computers, unfortunately, do what we tell them to do, not necessarily what we want them to do! There must be no ambiguity in the instructions that we give to a computer in our programmes, *i.e.*, no possibility of alternative interpretations. Since the computer always takes some course of action, great care must be taken to ensure that there is only one possible course of action so that we get the desired results. For instance, consider the statement: "Compute arithmetic mean for a given set of `n` observations". It seems to specify the computation we wish to get performed by computer. However, it is in fact too imprecise for computer to do the desired calculations, because too much detail is left unspecified, *e.g.*, where are the observed values, how many are there, are the missing/zero values to be included, and so on. This is the essence of computer programming.

Most interesting problems appear to be very complex from programming standpoint. For some problems such as mathematical problems, this complexity may be inherent in the problem itself. In many cases, however, it can be due to other factors that may be within our control, *e.g.*, incomplete or unclear specifications of the problem. In the development of computer programmes, complexity need not always be a problem if it can be properly controlled.

Computer programming can be difficult. However, we can do a great deal to make it easier. First, it is important that we separate the problem-solving phase of the task from implementation phase (Fig.1.1). In the former phase, we concentrate on the design of an algorithm to solve the stated problem. Only after ensuring that the formulated algorithm is appropriate, we translate the algorithm in some programming language such as 'C'. Given an algorithm that is precise, its translation into a computer programme is quite straightforward.

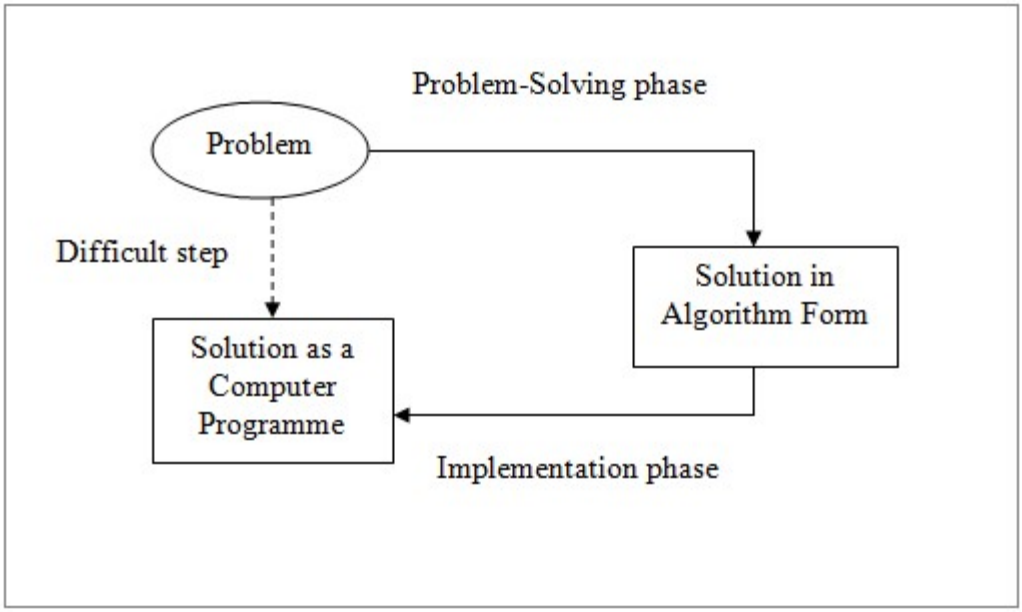


Fig. 1.1 Problem solving with computer

Hence, the essence of computer programming is the encoding of algorithms for subsequent execution on automatic computing machines. The notion of an algorithm is one of the basic ideas in mathematics and has been known and used since initial theories of Computer Science emerged. However, we can define the term algorithm in computer programming context as follows:

> *An algorithm is a list of instructions specifying a sequence of operations, which will give the answer to any problem of a given type.*

There are following six basic operations performed through a computer programme:

1. Input data
    a) Read data from a file
    b) Get data from the keyboard
2. Perform arithmetic computations using actual mathematical symbols or the words for the symbols
3. Assign a value to a variable
    There are three cases possible:
    a) Variable initialisation
    b) Assign a value to a variable as a result of some processing
    c) Keep a piece of information for later use (Save/Store)
4. Compare two pieces of information and select one of two (or more) alternative actions
5. Repeat a group of actions on the basis of some pre-defined condition or number of iterations
6. Output information
    a) Write information to a file
    b) Display information to the screen.

1.2 Properties of an Algorithm

Algorithms have several properties of interest: *a)* in general, the number of operations, which must be performed in solving a particular problem is not known beforehand; it depends on the particular problem and discovered only during the course of computation; *b)* the procedure specified by an algorithm is deterministic process, which can be repeated successfully at any time and by anyone; it must be given in the form of a finite list of instructions giving exact operation to be performed at each stage of the calculations; and *c)* the instructions comprising an algorithm define the task that may be carried out on any appropriate set of initial data and which, in each case, give the correct result. In other words, an algorithm tells how to solve not just one particular problem, but a whole class of similar problems.

1.3 Description of Algorithms

There are several approaches possible to the description of algorithms, *viz.*, narrative description, the flowchart, an algorithmic language, *etc*.

Narrative approach

It is a straightforward method of expressing an algorithm, *i.e.*, simply to outline its steps verbally. Recipes, for example, are normally expressed in this fashion. This is easy to do. However, natural language is wordy and imprecise and often quite unreliable as a vehicle for transferring information. Hence, natural language is unsuitable as the sole medium for the expression of algorithms. Despite its drawbacks, however, prose (*i.e.*, ordinary language without metrical structure) does have a place, possibly in presenting clarifying remarks or outlining special cases. Hence, it is not discarded entirely.

An illustration: Consider to design an algorithm to compute arithmetic mean for a given set of `n` observations using narrative approach.

1. Read the value of `n` (i.e., number of observations)
2. Read all then `n` observations one-by-one
3. Add these `n` observations `x1, x2, ...`
4. Divide the '`sum`' obtained in step-3, by `n` to calculate mean
5. Write the `mean` value obtained in step-4
6. Stop

The flowchart

It is often easier to convey ideas pictorially than verbally. Maps provide a convenient representation of the topology of a city, and may be of more value when you are lost than verbal directions. Assembly directions for a newly acquired piece of equipment are usually much more comprehensible if diagrams are included. A very early attempt at providing a pictorial format for the description of algorithms, which dates back to the days of von Neumann, involved the use of the flowchart. A flowchart shows the logic of an algorithm, emphasising the individual steps and their interconnections. Over the years, a relatively standard symbolism has emerged (see Table-1.1); *e.g.*, computations are represented by a rectangular shape, a decision by a diamond

shape, *etc*. The boxes are connected by directed lines indicating the order in which the operations are to be carried out.

Table 1.1 Flowchart symbols and description

Symbol	Name	Purpose
	Terminator	Marks beginning and end
	Data	Inputs data from input device
	Process	Depicts Operations such as initialisation of variables, assignments and mathematical calculations
	Decision	Branching control in more than one direction according to different conditions
	Display	Displays the programme output on output device
	Document	Writes results through printer (hard-copy)
	Connector	Connects two parts of a flowchart to form loop
	Directed lines	Depict the direction of flow of control

A simple flowchart for the above problem is shown in Fig.1.2.

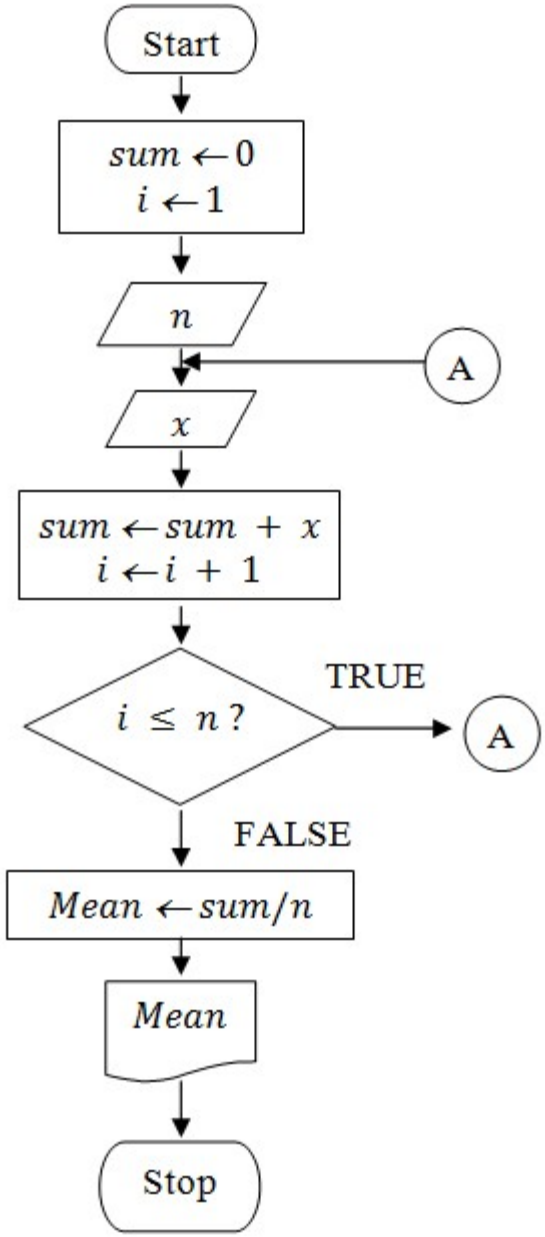


Fig. 1.2 Flowchart to compute average of *n* observations.

An algorithmic approach

Here, the approach is to extract the best features of the previous two approaches and combine these in a special language for the expression of algorithms. From the narrative approach, we borrow the descriptiveness of prose. To this, we add the conciseness of expression of the flowchart. The result is a kind of 'pidgin programming language', similar in flavour to several programming languages. We should try to avoid tying the algorithmic language too closely with any particular programming language.

Example1(a): Simple algorithm to compute average of *n* observations.

Let us try to design a simple algorithm using the algorithmic approach for the same example to compute average of `n` observations as follows.

1. [Initialise variables]
   `sum ← 0`
   `i←1`
2. [Input number of observations (n)]
   `Read (n)`
3. [Input n observations *(x)* one-by-one]
   `Read (x)`
4. [Add the x values one-by-one]
   `sum ← sum + x`
   `i ← i + 1`
5. [Check whether i ≤ n]
   `if i ≤ n`
   `then go to step-3`
   `else go to step-6`
6. [Compute mean value]
   `mean ← sum/n`
7. [Print the mean value]
   `Write('Mean =', mean)`
8. `Exit.`

Note that the above algorithm uses *goto* instructions to perform iteration. Edsger W. Dijkstra, the famous Dutch computer scientist who is well known for his contribution to the fields of algorithm design and programming languages, pointed out in a paper (Dijkstra, 1968) that the *goto* statement in programming languages should be abolished! This paper was written at a time when the conventional way of programming was to code iterative loops, *if ... then ... else* and other control structures by the programmer using *goto* statements as shown in the preceding algorithm. Most programming languages of that time did not support the basic control flow statements that we take for granted today; or merely provided very limited forms of the same. However, Dijkstra did not mean that all uses of *goto* were bad, but rather he highlighted the need for superior control structures that, when used properly, would eliminate most of the uses of *goto*, which was popular at that time. Dijkstra still allowed for the use of *goto* for more complicated programming control structures. Hence, a new paradigm called structured programming (to be discussed later) was emerged. The sophisticated structured programming languages like 'C' discourage the use of *goto* statements and rather provide proper control and looping structures such as *if ... then ... else* , *while ... do* loop, *report ... until* loop and *for* loop, which are discussed later (in perspective with 'C' programming) in greater details.

Now, let us revise the above algorithm so as to do the same task using a *while ... do* kind of looping mechanism (also called pre-test loop, as it tests for a condition prior to running a block of

code), which executes the underlying set of instructions as long as a pre-defined condition is fulfilled. If the condition is not TRUE at the onset, the loop even does not execute at all in such a case. Mind that *while ... do* loop is useful when you do not know exactly as to how much iteration the loop has to perform; for instance, in present example, the number of observations may vary for different datasets to be analysed by the programme.

Example 1(b): Revised algorithm to compute average of *n* numbers using *while ... do* loop.

1. [Initialise variables]
   sum ← 0
   i←1
2. [Input number of observations (n)]
   Read (n)
3. [set up a while loop, which continues unless i> n]
   while i ≤ n do
   [Input n observations (x) one-by-one]
   Read (*x*)
   [Add the *x* values one-by-one]
   sum ←sum + x
   i ← i + 1
   end while
4. [Compute mean value]
   mean ← sum/ n
5. [Print the mean value]
   Write('Mean =', mean)
6. Exit.

Further, let us modify the above algorithm so as to perform the same task using a *repeat ... until* kind of looping structure (also called post-test loop, as it tests the condition after running a block of code once), which executes the underlying set of instructions as long as a pre-defined condition is fulfilled. Even if the condition is not TRUE, the loop executes at least once in such a case. Note *repeat ... until* loop is useful like *while ... do* loop, when you do not know exactly as to how much iteration the loop has to perform; *e.g.*, in present example, the number of observations may vary for different datasets to be analysed by the programme.

Example 1(c): Revised algorithm to compute average of n numbers using *repeat ... until* loop.

1 [Initialise variables]
sum ←0
i←1
2 [Input number of observations (n)]
Read(n)

3 [set up a repeat…until loop, which continues as long as $i \leq$ n is TRUE]

do

[Input n observations (x) one-by-one]

Read(x)

[Add the x values one-by-one]

sum ← sum + x

i ← i + 1

while $i \leq$ n

4 [Compute mean value]

mean ← sum/ n

5 [Print the mean value]

Write('Mean =', mean)

6 Exit.

In case, it is known beforehand as to how much iteration the loop would exactly perform, you can make use of the for loop (also known as count loop). The purpose of for loop is to execute an initialisation, then keep executing a block of statements and updating an expression while a condition is TRUE. It is of the form:

```
for (initialization, condition, update)
        Block of statements
end for
```

This is illustrated with the following example.

Example 1(d): An algorithm to compute average of any ten numbers using for loop.

1. [Initialise variables]
   sum ← 0
2. [set up a 'for' loop, which iterates ten times]
   for i ← 1, $i \leq 10$, i ← i + 1 do
   [Input n observations (x) one-by-one]
   Read(x)
   [Add the x values one-by-one]
   sum ← sum + x
   end for
3. [Compute mean value]
   mean ← sum/10.0
4. [Print the mean value]
   Write('Mean =', mean)
5. Exit.

Note that here the variable *i* is not incremented after each iteration as the `for` statement automatically takes care of it. It performs exactly the specified iterations.

Moreover, there is another commonly used control structure called the selection structure. It represents the decision-making capabilities of the computer. This structure facilitates you to make a choice between available options. These choices are based on a decision about whether the respective condition is TRUE or FALSE.

There are many variations of the selection structure as follows:

1. Simple selection (simple `if` statement): applicable in case a choice is to be made between two alternatives. The keywords used are `if`, `then`, `else` and `endif`.
2. Simple selection with null false branch (null `else` statement): This is useful in case a task is performed only when a particular condition is true. The keywords used are `if`, `then` and `endif`..
3. Combined selection (combined `if` statement): This is used where the statement contains compound conditions. If the conditions are combined using the connector 'and', both conditions must be true for the combined condition to be true. If the connector 'or' is used to combine any two conditions, only one of the conditions needs to be true for the combined condition to be considered true. The keywords used are `if`, `then` and `endif`. Also, the 'not' operator is used for the logical negation of a condition and can be combined with the 'and' as well as 'or' operators. In order to avoid any ambiguity arising due to several conditions connected through 'and' or 'or' operators, better use parentheses to make the logic clear.
4. Nested selection (nested `if` statement): This is useful when the keyword `if` appears more than once with an `if` statement. It uses `if`, `then`, `else` and `endif` keywords. Nesting can be classified as linear or non-linear. The linear nested `if` statement is used when a field is being tested for various values and a difference action is taken for each value. Each `else` statement immediately follows its corresponding `if` condition. A non-linear nested `if` occurs when a number of different conditions need to be satisfied before a particular action is taken. An `else` statement can be separated from its corresponding `if` statement. Avoid using non-linear nested `if` statements as their logic is often hard to follow. Generally, indention is used for readability.

Example 2: An algorithm to input an examination's marks of *n* students as well as to test each student's marks for the award of a grade. The marks are assigned as a whole number in the range: 1 to 100. Grades are awarded according to the following criteria:

Serial Number	Criterion	Grade
1.	*marks >= 80*	Distinction
2.	*marks >= 60* but *< 80*	Merit
3.	*marks >= 40 but < 60*	Pass

4.	*marks < 40*	Fail

1. Read (n)
2. $i \leftarrow 1$
3. `while` $i \leq$ n do
   Read (marks for one student at a time)
   `if (marks ≥ 80)`
   `Write('Distinction')`
   `else if (marks ≥ 60 and marks < 80)`
   `Write('Merit')`
   `else if (marks ≥ 40 and marks <60)`
   `Write('Pass')`
   `else if (marks < 40)`
   `Write('Fail')`
   `end if`
   `i ← i + 1`
   `end while`
4. `Exit.`

Example 3: An algorithm to find and print the largest number among any *n* given numbers.

1. `Read(n, current_number)`
2. `max ← current_number`
3. `counter ← 1`
4. `while (counter <n) do`
   `counter ← counter +1`
   `Read (next_number)`
   `if (next_number > max) then`
   `max ← next_number`
   `end if`
   `end while`
5. `Write ('The largest number is :', max)`
6. `Exit.`

The above algorithms (Examples 2 and 3) nest *if ... then ... endif* structure within the *while ... do* loop structure. A care should be exercised while creating nested structures. The inner most structure should be properly embedded within the outer structure. Overlapping

between the two is not permissible; *e.g.*, in the above algorithm, `if` structure should not continue after the terminal statement of the *`while ... do`* loop, *i.e.*, *`while`* .

Lesson 2

PROBLEM SOLVING WITH COMPUTER PROGRAMMING – PART II
(Pseudo Codes and Analysis of Algorithms)

2.1 Pseudo Code

Pseudo code is a shorthand notation for programming, which uses a combination of informal programming structures and verbal descriptions of code. Emphasis is placed on expressing the behaviour or outcome of each portion of code rather than on strictly correct syntax.

In general, pseudo code is used to outline a programme before translating it into proper syntax. This helps in the initial planning of a programme, by creating the logical framework and sequence of the code. An additional benefit is that because pseudo code does not need to use a specific syntax, it can be translated into different programming languages and is, therefore, universal. It captures the logic and flow of a solution without the bulk of strict syntax rules.

It uses the structural conventions of a programming language, but is intended for human reading rather than machine reading. Pseudo code typically omits details that are not essential for human understanding of the algorithm, such as variable declarations, system-specific code and some subroutines. The programming language is augmented with natural language description details, where convenient, or with compact mathematical notation. The purpose of using pseudo code is that it is easier for people to understand than conventional programming language code, and that it is an efficient and environment-independent description of the key principles of an algorithm. It is commonly used in textbooks and scientific publications that are documenting various algorithms, and also in planning of computer programme development, for sketching out the structure of the programme before the actual coding takes place.

No standard for pseudo code syntax exists, as a programme in pseudo code is not an executable programme. Pseudo code resembles, but should not be confused with, skeleton programmes including dummy code, which can be compiled without errors. Flowcharts charts can be thought of as a graphical alternative to pseudo code.

While writing pseudo codes, it is important to keep in mind the precedence of operators (including arithmetic, relational and logical operators). Generally, expressions involving such operators are evaluated left-to-right following some precedence of operators just like the BODMAS (hierarchy of arithmetic operators) concept you used to follow while simplifying the algebraic expressions in your lower classes. For instance, consider the precedence of most commonly used arithmetic operators, *viz.*, addition, subtraction, multiplication and division. The addition and subtraction operators have same precedence, whereas multiplication and division operators have same precedence. However, level of precedence for multiplication and division operators is higher than that for the addition and subtraction operators. Hence, the following two expressions (on Right Hand Side (RHS) of '=' symbol) would evaluate to two different results:

$$grade = marks1 + marks2 + marks3 + marks4/4.0$$ and

$$grade = (marks1 + marks2 + marks3 + marks4)/4.0$$

Note that parentheses are used to alter the normal order of evaluation. A complete discussion is given later in perspective with the 'C' programming.

Example 1: Write an algorithm and a pseudo code as well as draw a flowchart to convert the length in feet to centimetre. Given that one foot contains 30.48 cm. (See Example-1 of Lesson-9 for the 'C' code corresponding to the following pseudo code).

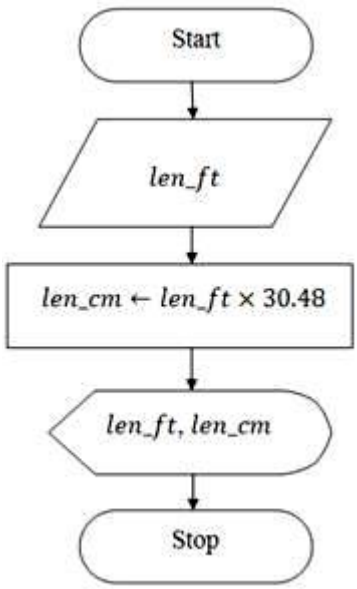
Algorithm

1. Read (len_ft)
2. $len\_cm \leftarrow len\_ft \times 30.48$
3. Write (len_ft, len_cm)
4. Exit.

Pseudo code

1. Use variables: len_ft and len_cm of the type floating-point
2. Input len_ft
3. $len\_cm = len\_ft \times 30.48$
4. Print len_ft, len_cm
5. End programme.

Flowchart



Example 2: Write an algorithm and draw a flowchart that will calculate the real roots of a quadratic equation. (See Example-2 of Lesson-9 for the 'C' code corresponding to the following pseudo code).

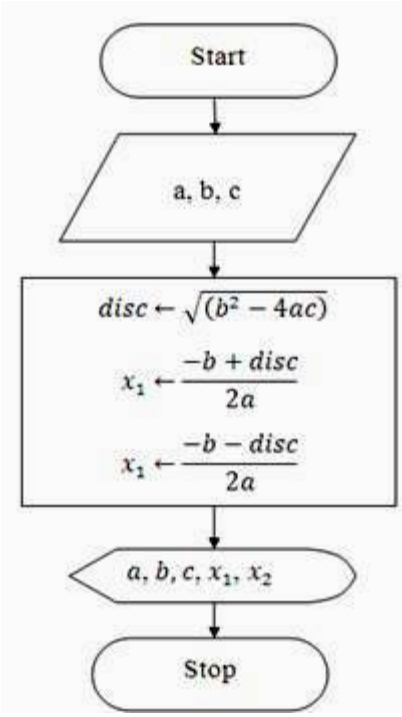
Algorithm

1. Read (a, b, c)
2. $d \leftarrow sqrt(b \times b - 4 \times a \times c)$
3. $x_1 \leftarrow (-b + d)/(2 \times a)$
4. $x_2 \leftarrow (-b - d)/(2 \times a)$
5. Write (a, b, c, x_1, x_2)
6. Exit.

Pseudo code

1. Use variables: a, b, c, d, x_1 and x_2 of type floating-point
2. Input values for a, b and c
3. $d = sqrt(b \times b - 4 \times a \times c)$
4. $x_1 = (-b + d)/(2 \times a)$
5. $x_2 = (-b - d)/(2 \times a)$
6. Print a, b, c, x_1, x_2
7. End programme.

Flowchart



Example 3: The following algorithm inputs a student's marks for four subjects; determines final grade by calculating the average of the four marks; and displays whether he/she is passing or failing on the basis that grade more than or equal to 50 is considered as 'pass', otherwise it is treated as 'fail'.

1. Read($marks1, marks2, marks3, marks4$)
2. $grade \leftarrow (marks1 + marks2 + marks3 + marks4)/4$
3. *if* $(grade < 50)$ *then*
   Write ("*FAIL*")
   else
   Write ("*PASS*")
   endif
4. Exit.

The students are advised to draw a flowchart and write a pseudo code for the above algorithm.

Example 4: Consider the algorithm developed in Example-3 of Lesson-1. Write a pseudo code and draw corresponding flowchart to find and print the largest number among any *n* given numbers.

1. Use the variables n, current_number, next_number, max and counter of the type integer.
2. Input values for n and current_number
3. max = current_number
4. counter =1
5. while condition *(counter < n)* holds good, do the following
   i) counter = counter +1
   ii) Input next_number
   iii) if (next_number>max) then
   max = next_number
   (Note: previous value of max *is overwritten);*
   else
   loop (*i.e.*, go to step-5)
   end if

end while

6. Print max with the preceding slogan, 'The largest number is :'

7. End programme.

The flowchart corresponding to the above pseudo code that finds and prints the largest number among any *n* given numbers is given below:

Example 5: Write a suitable pseudo code and draw flowchart corresponding to the algorithm given in Example-1(d) of Lesson-1 to compute average of any ten numbers using `for` loop.

Pseudo code

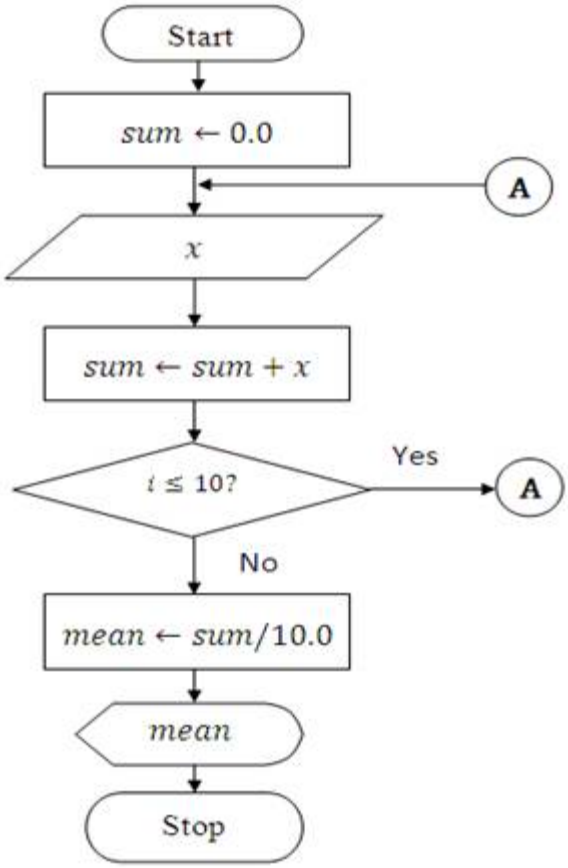
1. Use variable i of the type integer
2. Use variables x, $mean$ and sum of the type floating-point
3. Initialise the variable sum to the value, 0
4. for $(i = 1, i \leq 10, i = i + 1)$

   Input x

   $sum = sum + x$

   end for
5. $mean = sum/10.0$
6. Print $mean$
7. End programme.

Flowchart



Example 6: Pesudo code to sort given set of numbers in ascending order using the insertion sort method.

Suppose `x` is an array of `n` values. We want to sort `x` in ascending order. Insertion sort is an algorithm to do this. We traverse the array and insert each element into the sorted part of the list where it belongs. This usually involves pushing down the larger elements in the sorted part. The pseudo code segment is as follow:

```
INSERTION-SORT(A)
for (j = 2, j ≤ length(A), j = j + 1)
    key = A[j]
    i = j − 1
    while (i > 0 and A[i] > key)
      A[i + 1] = A[i]
      i = i − 1
    end while
    A[i + 1] = key
end for
```

In above pseudo code, the phrase **length (A)** denotes a way (which may be different for different programming languages; this may be even input by user at the run-time of the progrmme) of getting length of the array variable **A**. The 'C' function and complete programme for above pseudo code is given in Example-2 of Lesson-15.

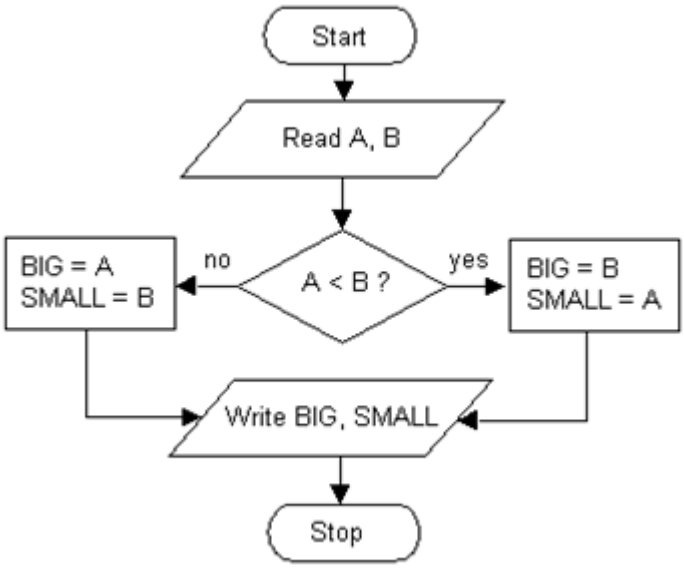
2.2 Analysis of Algorithms

The analysis of algorithms is the determination of the amount of resources (such as time and storage) necessary to execute them. Most algorithms are designed to work with inputs of arbitrary length. Usually the efficiency or running time of an algorithm is stated as a function relating the input length to the number of steps (*i.e.*, time complexity) or storage locations (*i.e.*, space complexity).

Algorithm analysis is an important part of a broader computational complexity theory, which provides theoretical estimates for the resources needed by any algorithm that solves a given computational problem. These estimates provide an insight into reasonable directions of search for efficient algorithms.

2.3 Practical Exercises for Lesson 2

1. The following flowchart depicts a programme that reads two given numbers; compares the two numbers; and finally, displays the numbers read in decreasing order:



Write an equivalent algorithm as well as pseudo code for the above flowchart.

2. Design an algorithm and the corresponding flowchart and pseudo code for adding the test scores as given below:

   26, 49, 98, 87, 62, 75

3. Write an algorithm and pseudo code, which accepts two numbers from the keyboard; calculates the sum and product of these numbers; and displays the results on the monitor's screen.

4. Write a flowchart and an algorithm that will output the square root of any positive number input from the keyboard until the number input is zero.
5. Design an algorithm and the corresponding pseudo code as well as flowchart for finding the sum of the numbers: 2,4,6,8…*n*.
6. Write an algorithm and draw a flowchart that will read the two sides of a rectangle and calculate its area.

Lesson 3

CONCEPTS OF PROGRAMMING LANGUAGE

3.1 Computer Programming Language

The design of modern computing machines parallels the algorithmic nature of most practical applications. The computer operates under the control of a series of instructions that reside in internal storage and are interpreted and executed by the circuitry of the arithmetic and control parts of the machine. The instructions are primitive in nature, each being composed basically of operation and one or more operands or modifiers, and exist in a form chosen for the internal representation of data. Machine instructions that exist in this form are said to be in machine language, since they are numerically coded (in binary codes) and directly executable by a specific computer.

Similarly as both an algorithm and a sequence of machine language instructions is the concept of a computer programme (usually referred to as a programme), which can be defined as follows:

> *A programme is a meaningful sequence of statements, possessing an implicit or explicit order of execution, and specifying a computer-oriented representation of an algorithmic process.*

Statements, in turn, are strings of symbols from a given alphabet, composed of letters, digits, and special characters. The forms of each statement obey a set of rules (syntax) and possess an operational meaning (semantics). Collectively, the alphabet, syntax and semantics are termed as a ‘language’.

Thus, a programming language is a standardised method for expressing instructions to a computer. The language allows a programmer to precisely specify what kinds of data a computer will act upon, and precisely what actions to take under various circumstances. This serves two primary purposes; the first is that the internal representation a computer uses for its data and operations (at the lowest level, *i.e.*, (0, 1) representation/machine language) are not easily understood by humans, so translating a human-readable language into those internal representations makes programming easier. Another purpose is transporting programmes between different computers: those internal representations also differ from one computer to the next, but if each is capable of translating the human-readable language into its own understandable language, then that programme will operate on both.

If the translation mechanism used, translates the programme text as a whole and then runs the internal format, this mechanism is called ‘compilation’. The compiler is, therefore, a programme, which takes the human-readable programme text (called source code or source programme) as data input and produces object code as output. This object code may be machine code directly usable by the processor, or it may be code matching the specification of a virtual machine, and run under that environment.

If the programme text is translated step by step at runtime, with each translated step being executed immediately, the translation mechanism is known as an interpreter. Interpreted programmes run usually more slowly than compiled programmes, but have more flexibility because, they are able to interact with the execution environment (*e.g.*, Disk Operating System – DOS environment), instead of all interactions being planned beforehand by the programmer.

Many languages can be either compiled or interpreted, but most are better suited for one than the other.

3.1.1 Low level languages

A low level programming language is a programming language that provides little or no abstraction from a computer's instruction set architecture. Generally, this refers to either machine code or assembly language. In present context, the word "low" indicates small or no amount of abstraction between the language and machine language. Thus, low level languages are sometimes described as being "close to the hardware".

Low Level Languages (LLL) can be converted to machine code without using a compiler or interpreter; and the resulting code runs directly on the processor. A programme written in a low level language runs faster as well as occupies a small amount of computer's memory as compared to an equivalent programme written in a high level language. Low level languages are simple, but are considered difficult to use, due to the numerous hardware details that the programmer must know prior to writing a code.

3.1.1.1 Machine language

Machine language is the programming language that the computer understands, *i.e.*, its native language. Machine language comprises the numeric codes for the operations that a particular computer can execute directly. These codes are strings of binary digits ("bits"), *i.e.*, 0's and 1's. Typically, machine language instructions use some bits to represent operations like addition or subtraction; while some of these bits represent operands or perhaps the location of the next instruction, *etc*. Machine language is difficult to read and write by the human beings as it does not resemble natural language or conventional mathematical notations. Moreover, machine language codes vary from computer to computer! Hence, machine languages instructions correspond to the instruction set of particular hardware architecture and are machine dependent, *i.e.*, different computers use different machine languages; however, every machine language supports instructions for basic operations such as addition and subtraction. The instructions are patterns of 0's and 1's in lengths of *16, 24, 32 or 64* bits. As stated earlier, each instruction has two parts: an operator and an operand. The operator is the first few bits of the instruction. It specifies the operation to be performed, such as movement of data between memory, the storage address and the Central Processing Unit (CPU). The remaining bits constitute the operand. The operand is usually an address of a memory location containing the data to be operated upon. Each address is 8-bit long and referred to as a word. Each machine language instruction is very simple and by

itself accomplishes very little. However, the CPU has the ability to execute millions of instructions per second.

Though machine language is the most efficient programming language in terms of execution, it has various drawbacks. Programming in machine language is tedious and error-prone. The use of numeric codes to represent instructions is difficult and makes programming a complex and arduous task. This complexity results in programmes that are difficult to read and even more difficult to debug. Besides, machine language requires the programmer to keep track of where individual bits of data are stored in the computer's memory. Even inserting new instructions or changing the address of one bit of data affects the correctness of the programme and the subsequent memory addresses used in the rest of the programme. Machine language programmes are very long as each instruction merely accomplishes a small task, *e.g.*, to multiply two numbers could require more than *10* instructions in machine language. These drawbacks make programming in machine language difficult and uneconomical. Such difficulties have given rise to the development of other languages.

3.1.1.2 Assembly language

Assembly language is one level above the machine language. It uses short mnemonic codes for instructions and allows the programmer to define names for blocks of memory that hold data. For instance, one might write a typical assembly language statement for an instruction that adds two numbers as "`ADD pay, total`" instead of the machine instruction like "`0110101100101000`".

Assembly language is designed to be easily translated into machine language. Although blocks of data may be referred to by name rather than their machine addresses, assembly language does not provide more sophisticated means of organising complex information like machine language; assembly language requires detailed knowledge of particular internal computer architecture. It is useful when such details are important, as in programming a computer to interact with input/output devices (*i.e.*, printers, scanners, storage devices, and so forth).

3.1.2 High level languages

High level programming languages allow the programmer to focus on the problem rather than concentrating on how the computer software and hardware interact to solve the problem. These programmes are designed for ease and reliability while machine language is designed for efficient execution. With the advent of High Level Languages (HLL), most programmers have very little use for machine language to write programmes. However, programmers still need to understand machine language. Knowledge of machine language is essential for the programmers who are responsible for maintaining existing machine language codes and creating assemblers. Besides, there are certain tasks that can be best accomplished through machine language.

3.2 Computer Programming Paradigms

3.2.1 Procedural programming

This is a method of programming based upon the concept of the unit and scope (the data viewing range of an executable code statement). A procedural programme is composed of one or more units or modules, either user coded or provided in a code library; each module is composed of one or more procedures, also called a function, routine, subroutine, or method, depending on programming language. It is possible for a procedural programme to have multiple levels or scopes with procedures defined inside other procedures. Each scope can contain variables, which cannot be seen in outer scopes.

Procedural programming offers many benefits over simple sequential programming: procedural programming code is easier to read and more maintainable; procedural code is more flexible; procedural programming allows for the easier practice of good programme design.

3.2.2 Structured programming

Structured programming is a programming paradigm that aims at improving the clarity, quality and development time of a computer programme by making extensive use of subroutines, block structures and `for` and `while` loops, in contrast to using simple tests and jumps such as the *goto* statement as stated earlier, which could lead to "spaghetti code" making it both difficult to follow and to maintain. This paradigm emerged way back in the 1960's and was bolstered theoretically by the Structured Programme Theorem, and practically by the emergence of languages such as ALGOL, 'C', Pascal, PL/1, *etc*. The structured programme theorem states that every computable function can be implemented in a programming language that combines sub-programmes in only three specific ways:

- Executing one sub-programme, and then another sub-programme (sequence);
- Executing one of two sub-programmes according to the value of a Boolean variable (selection); and
- Executing a sub-programme until a Boolean variable is true (repetition).

This can be seen as a subset or sub-discipline of procedural programming, one of the major paradigms (and probably the most popular one) for programming computers. It is possible to do structured programming with almost any procedural programming language. Hence, at the level of relatively small pieces of code, structured programming typically recommends simple, hierarchical programme flow structures, *viz.*, sequence, selection and repetition. These can be obtained in most modern languages by using only structured control (looping and branching) constructs, usually, named `while…do` `repeat … until`, `for`, and `if …then …else`. Often, it is recommended that each structured element should only have one entry point and one exit point, and a few languages enforce this.

Programmers should break larger pieces of code into shorter subroutines (functions and procedures in some languages) that are small enough to be understood easily. In general, programmes should use global variables carefully; instead, subroutines should use local variables and take arguments either by value or by reference (to be discussed later). These techniques help

to make isolated small pieces of code easier to understand without having to understand the whole programme at once.

Structured programming is often (but not always) associated with a top-down approach to design. In this way, designers map out the large scale structure of a programme in terms of smaller operations, implement and test the smaller operations, and then tie them together into a whole programme.

By the end of the 20th century, the vast majority of serious programmers endorsed structured procedural programming. They claim that a fellow can understand a structured programme more easily, leading to improved reliability and easier maintenance. Attempts to actually measure this have been rare, and there is a suspicion in some circles that the benefits are real but small. Further, designers have created new paradigms loosely based on procedural programming that accept the lessons of structured programming but attempt to go beyond this in providing structure for data as well as for programme flow. Object-Oriented Programming (OOP) in most cases can be seen as an example of this, although there are also some object-oriented variants that are not procedural.

You know that unstructured languages define control flow largely in terms of a `goto` statement that transfers execution control to a label in code. Structured programming languages provide special constructs for creating a variety of loops and conditional branches of execution, although they may also provide a `goto` statement so as to reduce excessive nesting of cascades of `if` structures, especially for handling exceptional conditions.

To be more precise, any code structure should have only one entry point and one exit point in a structured programming language. Many languages such as 'C', allow multiple paths to a structure's exit (such as `continue`, `break` and `return`) which can bring advantages in readability.

3.2.3 Object-oriented programming

It is a software design methodology that defines programmes in terms of 'objects', which are entities that combine both state (*i.e.*, data) and behaviour (*i.e.*, procedures or methods). Object-oriented programming expresses a programme as a set of these objects, which communicate with each other to perform tasks. This differs from traditional procedural languages in which data and procedures are separate and unrelated. These methods are intended to make programmes and modules easier to write, maintain and reuse.

Another way this is often expressed is that object-oriented programming encourages the programmer to think of programmes primarily in terms of the data types, and secondarily on the operations ('methods') specific to those data types. Procedural languages encourage the programmer to think primarily in terms of procedures, and secondarily the data that those procedures operate on. Procedural programmers write functions, and then pass data to them. Object-oriented programmers define objects with data and methods then send messages to the objects telling them to perform those methods on themselves.

The following four features are most important for an OOP language:

- **Abstraction:** Each object in the system serves as a model of an abstract 'actor' that can perform work, report on and change its state, and 'communicate' with other objects in the system, without revealing how these features are implemented. Processes, functions or methods may also be so abstracted, and when they are, a variety of techniques are required to extend an abstraction.
- **Encapsulation:** Also called 'information hiding', this ensures that objects cannot change the internal state of other objects in unexpected ways; only the object's own internal methods are allowed to access its state. Each type of object exposes an interface to other objects that specifies how other objects may interact with it. Some languages relax this, allowing some direct access to object internals in a controlled way, and limiting the degree of abstraction.
- **Polymorphism:** References to and collections of objects may contain objects of different types, and invoking a behaviour on a reference will produce the correct behaviour for the actual type of the referent. When it occurs at "run time", this latter feature is called 'late binding' or 'dynamic binding'. Some languages provide more static (compile time) means of polymorphism such as C++ templates and operator overloading.
- **Inheritance:** Organises and facilitates polymorphism and encapsulation by permitting objects to be defined and created that are specialised types of already-existing objects - these can share (and extend) their behaviour without having to re-implement that behaviour. This is typically done by grouping objects into 'classes', and classes into 'trees' or 'lattices' reflecting common behaviour.

Just as procedural programming led to refinements of technique such as structured programming, modern object-oriented software design methods include refinements such as the use of design patterns, design by contract, and modelling languages (such as UML).

3.3 Programming Languages for Scientific and Business Computing

FORTRAN

The name 'Fortran' is an abbreviation of the phrase '**For**mula **Tran**slation'. The language was originally known as FORTRAN. With the advent of 'Fortran 90', the capitalisation has been abandoned. The published formal standards use "Fortran".

This programming language was developed in 1950s by a team lead by John W. Backus whilst he was at IBM; and it is still in use today! This compiler was the first compiler for any HLL, and was actually an optimising compiler, because the authors were worried that no one would use the language if its performance was not comparable to assembly language.

It is a procedural language mainly used for scientific computing and numerical analysis. Owing to the heavy use by scientists involved in numerical work, the language grew in ways that encouraged compiler writers to produce compilers that generated high quality (efficient) code. There are many high performance compiler vendors. Much work and research in compiler theory and design was motivated by the need to generate good code for Fortran programmes.

Several revisions of the language have appeared, including the very well known FORTRAN IV (the same as FORTRAN 66), FORTRAN 77 and Fortran 95. The most recent formal standard for the language is known as FORTRAN 2003.

Initially, the language relied on precise formatting of the source code and heavy use of statement numbers and `GOTO` statements. Every version introduced 'modern' programming concepts such as source code comments and output of text; IF-THEN-ELSE (in FORTRAN 77); recursion (in Fortran 95); *etc*. Of late, Fortran 2003 supports several new feature as exception handling; allocatable components and dummy arguments; interoperability with 'C'; object-orientation including procedure pointers and structure components, structure finalisation, type extension and inheritance; and polymorphism.

Vendors of high performance scientific computers such as Burroughs, CDC, CRAY, IBM, Texas Instruments, *etc*., added extensions to FORTRAN to make use of special hardware features like instruction cache, CPU pipeline, vector arrays, *etc*. For example, one of IBM's FORTRAN compilers had a level of optimisation, which reordered the machine code instructions to keep several internal arithmetic units busy at a time. Another example is the special 'version' of FORTRAN designed specifically for the ILLIAC IV supercomputer installed at the NASA's Ames Research Centre. All such extensions have disappeared with the passage of time. However, a few major extensions as OpenMP and CoArray Fortran are operational, which facilitate shared memory programming and parallel programming, respectively.

BASIC

BASIC is a general-purpose HLL whose design philosophy emphasises ease of use. Its name is an acronym from **B**eginner's **A**ll-purpose **S**ymbolic **I**nstruction **C**ode. The BASIC was originally designed by John George Kemeny and Thomas Eugene Kurtz in 1964 at Dartmouth College in New Hampshire, USA to provide computer access to non-science students as the use of computers required writing custom software at that time; which restricted computers' use only to the scientists and mathematicians. It is an interpreter-based language. The language and its variants became widespread on microcomputers in the late 1970s and 1980s.

BASIC remains popular in numerous dialects and new languages influenced by BASIC such as Microsoft Visual Basic. A large group of developers for the .NET framework uses Visual Basic .NET as their programming language.

COBOL

COBOL is an HLL developed by the **Co**nference on **Da**ta **Sy**stems **L**anguages (CODASYL) Committee in 1960. Subsequently, the responsibility for developing new COBOL standards has been assumed by the American National Standards Institute (ANSI). Three ANSI standards for COBOL have been produced in 1968, 1974 and 1985.

The word **COBOL** is an acronym that stands for **CO**mmon **B**usiness **O**riented **L**anguage. As the expanded acronym indicates, COBOL is designed for developing business, typically file-oriented,

applications. It is not designed for writing systems programmes. For instance, you would not develop an operating system or a compiler using COBOL.

Conventionally, COBOL is a simple language with a limited scope of function having no pointers, no user-defined functions and no user-defined types. It encourages a simple straightforward programming style. Despite these limitations, COBOL has proven itself to be well suited to business computing. Most COBOL programmes operate in a domain where the programme complexity lies in the business rules that have to be encoded rather than in the sophistication of the data structures or algorithms required.

Now, the Object-Oriented COBOL (OO-COBOL) has been developed that retains all the advantages of previous versions but now includes: user-defined functions; object orientation; national characters – Unicode; multiple currency symbols; cultural adaptability (locales); dynamic memory allocation (pointers); data validation; binary and floating point data types; user-defined data types. Hence, using these OO-COBOL constructs one can write well structured programmes.

COBOL is non-proprietary, *i.e.*, its standard does not belong to any particular vendor. The ANSI COBOL committee legislates formal, non-vendor-specific syntax and semantic language standards.

A typical COBOL programme comprises the four divisions, which divide it into distinct structural elements. Although some of the divisions may be omitted, the sequence in which they are specified is fixed, and must follow the order below:

- `IDENTIFICATION DIVISION.` Contains programme information
- `ENVIRONMENT DIVISION.` Contains environment information
- `DATA DIVISION.` Contains data descriptions
- `PROCEDURE DIVISION.` Contains the programme algorithms.

C

'C' programming language (called 'C' now onwards) is a general purpose procedural computer programming language developed in 1972 by Dennis Ritchie at the AT&T Bell Laboratories, New Jersey (USA) for use with the UNIX operating system. It is one of the most widely used programming languages especially for writing system software such as operating systems and compilers. Besides, it is commonly used for writing a wide variety of scientific and industrial applications including process automation for the dairy plants and dairy farms via embedded microcontrollers, Programmable Logic Controller (PLC), *etc*. 'C' has greatly influenced many other popular programming languages, most notably C++, which began as an extension to 'C'.

'C' is a procedural language for systems implementation. It was designed to be compiled using a relatively straightforward compiler to provide low-level access to memory, to provide language constructs that map efficiently to machine instructions, and to require minimal run-time support. Thus, 'C' was useful for many applications that had formerly been coded in assembly language.

Besides its low-level capabilities, the language was designed to encourage cross-platform programming. A standards-compliant and portably written 'C' programme can be compiled for a very wide variety of computer platforms and operating systems with few changes to its source code. The language has become available on a large range of platforms, from embedded microcontrollers to supercomputers.

Similarly as most procedural languages, 'C' facilitates structured programming as well as allows lexical variable scope and recursion. In 'C', the entire executable code is embedded in subroutines know as 'functions'. Function parameters are always passed by value. Pass-by-reference is simulated in 'C' by explicitly passing pointer values. 'C' source programme's text is free-format, using the semicolon as a statement terminator and curly braces for grouping blocks of statements.

C++

C++ (pronounced 'see plus plus') is an OOP language based on 'C', developed by Bjarne Stroustrup at Bell Labs in the 1980s. The name of the language written as C++ signifies the fact that the language evolved from 'C'. This name was suggested by Rick Mascitti in 1983. Earlier the language had been referred to simply as 'C with Classes'. The C++ language as an enhancement to the 'C' language introduced many novel features such as classes, virtual functions, operator overloading, multiple inheritance, templates and exception handling, *etc*. The C++ language was standardised by ISO as ISO/IEC 14882:1998: Programming Language C++ Standard way back in September, 1998. The standard further evolved with the standardisation of language and library extensions. The current standard for C++ programming language was ratified and published by ISO in September, 2011 as ISO/IEC 14882:2011 (informally known as C++11).

C++ is one of the most popular programming languages with application domains including systems software; application software; device drivers; embedded software; high-performance server and client applications; entertainment software such as video games; and hardware design. Several groups provide both free and proprietary C++ compiler software including the GNU Project, Microsoft, Intel, *etc*. C++ has greatly influenced many other popular programming languages, mainly C# (pronounced as 'see sharp') and Java.

Java

Java is an HLL like 'C' and Visual Basic, but Java compilers differ from most other compilers in that they produce compiled binary instructions not for the target computer hardware machine, but for an abstract computer platform called the Java Virtual Machine (JVM). Actually, JVM does not exist in hardware. Instead, the binary instructions are executed not by the processor of a computer, but by a computer programme that emulates this theoretical machine. For instance, while the binary code generated by the C++ language compiler will vary for Intel-processor based computers, Macintosh computers and Solaris based computers, the binary codes produced by the Java compiler are the same on all the three platforms. The programme that interprets those binary statements is the only thing that varies. Hence, Java facilitates programmers to write a programme and compile it on any platform, and feel confident that the compiled code runs without change on

any other computer platform. This is the promise of "write once, run anywhere," that makes Java such a persuasive language for application development. Of course, this portability comes at a price. You cannot expect a programme, which interprets the binary instructions to run as fast as an actual computer that executes these instructions directly, *i.e.*, Java programmes run relatively slower than the programmes written in compiled languages. However, there have been a number of techniques applied to Java interpreters to make them more competitive in speed. Java Just-In-Time (JIT) compilers are the general solution to this problem. These compilers identify the spots in a programme while it is running that can benefit by being optimised and carefully restructure the code or compile little pieces of it into machine language as it is executed so that if that segment is run repetitively its speed will approach that of compiled languages. The first significant JIT compilers were from Symantec (now Webgain) and Microsoft. Microsoft seems to have abandoned this work, but the Symantec JIT compiler is widely available. More recently, both Sun and IBM have introduced JIT compiler systems, which are quite powerful. Sun's compiler is referred to as its Hot Spot compiler and is provided with the Windows and Solaris versions of Java 1.3. The primary characteristics of Java language are as follows:

- Java is a true object-oriented language. It uses the principles of abstraction, encapsulation and inheritance. It is largely based on the notion of classes.
- Java language makes it very easy to develop concurrent processes, which execute simultaneously within an application. This is called multithreading. The multithread mechanism provides rich functionality to applications. For example, a user may need to fill in a form while printing a document. Also, multithreading enables Java function on multi-processor machines. Though some other languages provide the multithreading mechanism, but they use non-standard external libraries to accomplish it. Java integrates this capability directly into the language, which provides greater portability with Java programmes.
- Links are not edited in Java. Link editing is the step after compilation to link external libraries. In Java, the existence of libraries is verified during compilation and code is loaded from these libraries when the programme is executed. This decreases the size of executable codes and makes it possible to optimise the loading of libraries. Any reference to a library requires that it be accessible at the time of compilation. However, the most significant contribution of dynamic linking is that it allows the creation of applications that can be extended dynamically by simply adding new classes, without requiring possession of the source code.
- Java has a garbage collector, which is a mechanism that clears memory of all objects that are no longer being used. In 'C' and C++ programmes, developers must handle the destruction of created objects themselves. This method of managing memory in these programmes requires many lines of code and a lot of fine-tuning. The pointer concept no longer exists in Java.
- All Java applications run in a secure environment. The virtual machine performs very strict verification of Java code before it is executed. Code cannot bypass the protection mechanisms imposed by the language. For example, a class cannot access a field of another class that has been declared private. The code also cannot try to define pointers to directly access memory.

Higher-level verifications are performed by browsers, *e.g.*, an applet cannot access machine resources.

- The syntax of Java language is simplified and based on C++. The absence of pointers and memory management through the garbage collector as well as the requirement that all developments must use objects, make application code much easier to read. However, developers must yet understand and integrate the hierarchy of the object model of Java Application Programming Interfaces (API).
- Portability is the most argued topic when discussing Java. In reality, this characteristic is not related to the language itself, but rather to the runtime environment (the virtual machine) of Java programmes. There are currently virtual machines for Unix, Linux, Windows and Macintosh platforms. Nevertheless, the only guarantee for portability in an application is effective testing because problems can still occur between the different platforms.
- The Sun product distinguishes between two types of platforms. The first is the Java Base Platform, which improves as new API are published and integrated by Sun. The second is the Java Embedded Platform. This is a specialised embedded platform for terminals or peripherals with a minimal graphical interface (in fact, no graphical interface) and is executed using less than 0.5 MB. The Java Base Platform is a complex construction of Java libraries and classes. The key element of this platform is the runtime environment of Java applications, the virtual machine. This is the 'system' layer, which communicates with the operating system and provides application portability. The Java virtual machine is actually a runtime environment that interprets resource files (bytecode) or p-code in its own environment.

3.3 'C' Programme Compilation

'C' programming language is a free-form language, *i.e.*, the 'C' compiler allows you to write a statement anywhere on the line. Generally, 'C' statements are written in lowercase letters. However, uppercase letters are used for symbolic constants (to be discussed later).

The Borland Turbo C++ 3.0 compiler is used in this e-course. You may freely download this software from the World Wide Web (WWW) and install it on the appropriate hard drive and folder (say, on the D:\TC) of your computer system. After successful installation, you will find several sub-folders existing under the folder 'TC' on D drive. The executable command to start the C++ software is, TC.EXE, which is available under the sub-folder D:\TC\BIN. Hence, initiate the C++ software by running D:\TC\BIN\TC.EXE command at START►RUN under MS-Windows operating system.

Now, running (or executing) a 'C' programme involves the following sequence of steps:

i) Creating the programme
ii) Compiling the programme
iii) Linking the programme with functions that are needed from the 'C' library
iv) Executing the programme.

For illustration, consider the following simple 'C' programme that displays the slogan 'I Love My India!' onto the computer monitor.

```
/*
* File: India.c
* -------------
* This simple C programme prints out the slogan, "I Love My
India!".
*/
#include <stdio.h>
void main
{
    printf("I Love My India!\n");
}
```

You may write this 'C' programme using the Notepad and save it with the extension as `.c` rather than the Notepad default extension, `.txt` (say, `India.c`). Alternatively, you may write the same code using the in-built editor provided by many compilers such as the Borland Turbo C++ 3.0 compiler.

Run the C++ software as described above and compile the programme, `India.c` by clicking on the 'Compile' option on the toolbar and further selecting 'Compile' sub-option. If the programme is grammatically (or syntactically) correct, the message – "Success: Press any key" is flashed by the compiler, otherwise several errors may appear at the bottom, which must be corrected in order to execute the code.

After successful compilation of the programme, click on the 'Run' option on the toolbar to execute the programme. As a result, the slogan, "`I Love My India!`" appears on the computer monitor, which can be viewed by clicking on the option 'File' on the toolbar and selecting the 'DOS Shell' sub-option. After viewing the slogan, type the command 'exit' at the DOS prompt to come back to the C++ software. You may quit the C++ software and come back to MS-Windows operating environment by clicking toolbar-option 'File' followed by sub-option 'Quit'.

Lesson 4

CHARACTERISTICS OF 'C' LANGUAGE

4.1 Introduction

The main characteristics of 'C' include modularity, portability, extendibility, speed and flexibility. Modularity is the ability to split a larger programme into manageable small segments called modules. This is an important feature of structured programming languages. It helps to complete the software development projects in time as well as makes debugging process easier and quick. The codes written in 'C' are highly portable, *i.e.*, it is possible to install the software developed in 'C' on different platforms with minimum (or no) alterations in the source code. 'C' allows extending the existing software by adding new features to it. 'C' is also known as LLL because 'C' codes run at the speeds matching to that of the same programmes written in assembly language. That is, 'C' has both the merits of HLL and LLL. Thus, 'C' is mainly used in developing system software such as operating systems, *e.g.*, UNIX was written in 'C'. 'C' has right number of reserved words or special words also called keywords (ANSI 'C' has 32 keywords), which allow the programmers to enjoy flexibility and to have complete control on the language. Hence, 'C' is known as programmer's language as it facilitates them to induce creativity into their programmes.

4.2 Character Set

The character set in 'C' can be grouped into four major categories, *viz.*, letters, digits, special characters and white spaces. The 'C' character set comprises of digits: `0-9`; uppercase letters: `A-Z` and lowercase letters: `a-z`. The set of special characters supported by 'C' is given in Table-4.1.

Table 4.1 'C' Special Character Set

Serial Number	Character	Description
1.	,	Comma
2.	.	Period
3.	;	Semicolon
4.	:	Colon
5.	?	Question mark
6.	'	Apostrophe
7.	"	Quotation marks
8.	!	Exclamation mark
9.	|	Vertical bar
10.	/	Slash
11.	\	Backslash
12.	~	Tilde
13.	_	Underscore

14.	=	Equal sign
15.	%	Percentage sign
16.	&	Ampersand
17.	^	Caret
18.	*	Asterisk
19.	-	Minus sign
20.	+	Plus sign
21.	<	Opening angle ('less than' sign)
22.	>	Closing angle ('greater than' sign)
23.	(	Left parenthesis
24.	)	Right parenthesis
25.	[	Left bracket
26.	]	Right bracket
27.	{	Left Brace
28.	}	Right brace
29.	#	Number sign

Most versions of 'C' allow certain other characters such as $ and @ to be included within strings and comment statements. 'C' also uses certain combinations of these characters like \b, \n and \t to represent special conditions called backspace, newline and horizontal tab, respectively. Typically, these character combinations are known as escape sequences (*to be discussed later*). At this juncture, it is just to mention for completeness that each escape sequence represents a single character even though it is written as combination of two or more characters.

White space characters include blank space, horizontal tab, vertical tab, carriage return, newline and form feed. All these characters are known as white space characters because they serve the same purpose as the spaces between words and lines on a printed page so as to make the programme-reading process easier for human comprehension. White spaces are ignored by the compiler until they are a part of string (alphanumeric or non-numeric) constant. White space may be used to separate words, but are strictly prohibited while using between characters of keywords or identifiers.

4.3 Tokens

The tokens of a language are the basic building blocks, which can be assembled in a systematic order to construct a programme. Thus, in a 'C' source programme, the basic element recognised by the compiler is the "token". A token is text contained in the source programme that the compiler does not break down further into component elements, *i.e.*, tokens are the smallest individual words, punctuation marks, *etc*., recognised by the 'C' compiler. 'C' has six such types of token, *viz.*, keyword; identifier; constants and literals; operator and punctuator.

Instructions in 'C' are formed using syntax and keywords. It is necessary to strictly follow 'C' syntax rules, *i.e.*, all programmes must conform to grammatical rules pre-defined in the language. Any instruction that mismatches with the prescribed syntax encounters an error while compiling the programme. Each keyword (or reserved word) in 'C' has its own pre-defined meaning and relevance; hence, keywords should neither be used as variables nor as constant names. A list of 'C' keywords is given in Table-4.2 below.

Table 4.2 'C' Keywords

Serial Number	Keyword	Description
1.	auto	The default storage class.
2.	break	Command that exits for, while, switch, and do...while statements unconditionally.
3.	case	Command used within the switch statement.
4.	char	The simplest 'C' data type.
5.	const	Data modifier that prevents a variable from being changed. Also, see volatile.
6.	continue	Command that resets for, while, or do...while loop statement to the next iteration.
7.	default	Command used within the switch statement to catch any instances not specified with a case statement.
8.	do	Looping command used in conjunction with the while statement. The loop will always execute at least once.
9.	double	Data type that can hold double-precision floating-point values.
10.	else	Statement signalling alternative statements to be executed when condition underlying if statement evaluates as FALSE.
11.	enum	Data type that allows variables to be declared that accept only certain values.
12.	extern	Data modifier indicating that a variable will be declared in another area of the programme.
13.	float	Data type used for floating-point numbers.
14.	for	Looping command that contains initialisation, incrementation and conditional sections.
15.	goto	Command that causes a jump to a pre-defined label.
16.	if	Command used to change programme flow based on a TRUE/FALSE decision.
17.	int	Data type used to hold integer values.
	long	Data type used to hold larger integer values than int.

18.

19.	register	Storage modifier that specifies that a variable should be stored in a register if possible.
20.	return	Command that causes programme flow to exit from the current function and return to the calling function. It can also be used to return a single value.
21.	short	Data type used to hold integers. It isn't commonly used, and it's the same size as an int on most of the machines.
22.	signed	Modifier used to signify that a variable can have both positive and negative values. Also, see unsigned.
23.	sizeof	Operator that returns the size of the item in bytes.
24.	static	Modifier used to signify that the compiler should retain the variable's value.
25.	struct	Keyword used to group 'C' variables of any data type together.
26.	switch	Statement used to change programme flow in various directions. It is used in combination with the case statement.
27.	typedef	Modifier used to create new names for existing variable and function types.
28.	union	Keyword used to allow multiple variables to share the same memory space.
29.	unsigned	Modifier used to signify that a variable will contain only positive values. Also, see signed.
30.	void	Keyword used to signify either that a function doesn't return anything or that a pointer being used is considered generic or able to point to any data type.
31.	volatile	Modifier that signifies that a variable can be changed. Also, see const.
32.	while	Looping statement that executes a section of code as long as a condition remains TRUE.

Also, some compilers may include some or all of the following keywords:

`ada`	`Far`	`Near`
`asm`	`fortran`	`pascal`
`entry`	`huge`	

The list of keywords supported by C++ programming language in addition to the above noted keywords is also given here under. These C++ reserved words aren't within the scope of this

e-course; but in due course it is quite possible that you may like to port your 'C' programmes to C++, therefore, you should avoid using these keywords as well in your 'C' programmes.

catch	inline	Template
class	new	This
delete	operator	Throw
except	private	Try
finally	protected	Virtual
friend	public	

4.4 Identifiers

The term identifier refers to the name of various programme elements, *i.e.*, a variable, a function, an array, *etc*. Both, uppercase as well as lowercase letters are permitted; although, common practice is to use lowercase letters. The underscore character is also permitted in identifiers. It is usually used as a link between two words in an identifier having a lengthy name, *e.g.*, `fat_content`. The syntactic rules for assigning a name to an identifier are:

- The first character must be a letter or an underscore
- It must consist of only letters, digits or underscore
- First 31 characters are significant, *i.e.*, if the programmer writes an identifier comprising of more than 31 characters, the compiler considers only first 31 characters and ignores the remaining characters *(note that some implementations of 'C' recognise only first eight characters)*
- A keyword cannot be used as an identifier name
- It cannot contain any spaces.

Examples

The following names are valid identifiers:

X	y12	trial_1	_temperature
specie	sensory_score	win_tmp	TABLE

The following names are invalid identifiers for the reasons stated against each:

4th	:	First character must be a letter
"x"	:	Illegal character (")
order-number	:	Illegal character (-)
viscosity score	:	Illegal character (blank space).

Note: All the keywords are lowercase. Since uppercase and lowercase characters are treated as different by the 'C' compiler, it is possible to use uppercase keyword as an identifier. Generally, this is not done and considered as a poor programming practice. For instance, the keyword '`double`' and the identifier '`DOUBLE`' are different

4.5 Constants

Constants in 'C' refer to the fixed values that do not change during the execution of the programme. There are four basic types of constant in 'C', *viz.*, integer constants; floating-point constants; character constants and string constants. Besides, there are enumeration constants (to be discussed later). The integer and floating-point constants are collectively known as numeric constants. The rules governing all numeric constants are as follows:

- Commas and blank spaces cannot be included within the constant
- The constant can be preceded by a minus (-) sign, if needed
- The value of a constant cannot exceed specified minimum and maximum bounds.
- For each type of constant, these bounds vary from compiler to compiler.

Integer constants

An integer constant is an integer-valued number. Thus, it comprises of a sequence of digits. Integer constants can be written in three different systems, *viz.*, decimal (base 10), octal (base 8) and hexadecimal (base 16) systems. The novice programmers, generally, use decimal integer constants. A decimal integer constant may consist of any combination of digits taken from the set `0` through `9`. If the constant contains two or more digits, the first digit must be other than `0`.

Examples

Several valid constants are given below:

```
0          1          786        9797       32767      9999
```

The following integer constants are incorrect for the reasons stated against each:

12,456	Illegal character (,)
36.0	Illegal character (.)
10 20 30	Illegal character (blank spaces)
123 – 45 – 6780	Illegal character (–)
0600	The first digit cannot be a zero.

Floating-point constants

A floating-point (or real) constant is a base-10 number that contains either a decimal or an exponent (or both), *i.e.*, the floating-point constants are of two forms, *viz.*, fractional form and the exponential form. A floating-point constant in fractional form must have:

- At least one digit
- A decimal point
- Positive or negative sign (default sign is positive)
- No commas or spaces embedded in it.

A floating-point constant in its fractional form use four bytes in memory. For example, -26.9876, +867.9 and 654.0 are valid floating-point constants.

In exponential form, the floating-point constant is represented in two parts, *e.g.*, the floating-point constant, 0.00032 is written in exponential form as +3.2e -4. The part preceding the 'e' is known as 'mantissa'; and the other succeeding 'e' is called 'exponent'. A floating-point constant in exponential form must follow the rules:

- The mantissa part and the exponential part should be separated by the letter 'e'
- The mantissa may have a positive or negative sign (default sign is positive)
- The exponent must have at least one digit
- The exponent may be a positive or negative integer (default sign is positive)

Floating-point constants encompass a much greater range than integer constants. Typically, the magnitude of a floating-point constant might range from a minimum value of approximately 3.4 e -38 to a maximum of 3.4e +38 . Some versions of the language allow floating-point constants that cover a wider range such as 1.7e - 308 to 1.7e +308. Note that the value 0.0 (even being less than 3.4e -38 and 1.7e - 308) is a valid floating-point constant. You should find out the suitable values for the version of 'C' used on your computer system.

Examples of some valid floating-point constants:

0.	1.	0.2	627.602	267.0066	10000.
$3e-5$	0.000786	16.2	$1.9666E+8$	.21212121e12	$.002e-3$

Examples of some invalid floating-point constants *(the reasons specified against each)*:

2	Either a decimal point or an exponent must be present
7,000.0	Illegal character (,)
$6e+10.2$	The exponent must be an integer quantity (it cannot contain a decimal point)
2e 10	Illegal character (blank space) in the exponent.

Floating-point constants are normally represented as 'double-precision' quantities. Thus, each floating-point constant occupies eight bytes of memory. Some versions of 'C' allow the specification of a 'single-precision' floating-point constant by suffixing the letter 'F' (in either uppercase or lowercase) to the end of the constant, *e.g.*, 3e5F. Similarly, some versions of 'C' permit the specification of a 'long' floating-point constant by appending the letter 'L' (uppercase or lowercase), *e.g.*, 0.987654321e – 22L. Note that precision of floating-point constants, *i.e.*, the number of significant figures is different for different versions of 'C'. Basically, all these versions permit at least six significant figures and some versions permit as many as eighteen significant figures. You should determine the appropriate number of significant figures for your particular version of 'C'.

Character constants

A character constant is an alphabet, a single digit or a single special character enclosed within single quotation marks. The maximum length of a character constant can be 1 character. It occupies one byte of memory.

Examples

```
'A'          'x'          '3'          ' '
```

Note that the last constant consists of a blank space enclosed in quotation marks.

Character constants have integer values that are determined by the machine's specific character set. Thus, the value of a character constant may vary from computer to computer. However, the constants themselves are independent of the character set. Majority of computer systems and virtually all personal computers use American Standard Code for Information Interchange (ASCII) character set (visit http://blob.perl.org/books/beginning-perl/3145_AppF.pdf for further details), in which each individual character is numerically encoded with a unique 7-bit combination (hence, a total of 2^7 =128 differenr characters).

Example: several character constants and their corresponding values as per the ASCII character set are shown below:

Constant	Value	Constant	Value	Constant	Value
'A'	65	'x'	120	'3'	51
'?'	63	' '	32		

Note that these values will be the same for all the computers using the ASCII character set. However, the values will be different for computers using an alternate character set such as IBM mainframe computers used EBCDIC character set, which is based on its own unique 8-bit combination.

Escape sequences

Certain non-printing characters as well as the backslash and the apostrophe (') can be expressed in terms of escape sequences. An escape sequence always begins with a backward slash (\) followed by one or more special characters, *e.g.*, a line feed (called newline in 'C' perspective) is represented as `\n`. Such escape sequences always represent single characters, even though they comprise of two or more characters. The commonly used escape sequences along with their corresponding ASCII values are listed in Table-4.3.

Table 4.3 Some Escape Sequences *vis-à-vis* ASCII Values

Character	Escape Sequence	ASCII Value
null	\0	000
bell	\a	007
backspace	\b	008
horizontal tab	\t	009
newline	\n	010
vertical tab	\v	011
form feed	\f	012
carriage return	\r	013
quotation mark	\"	034

apostrophe	\'	039
question mark	\?	063
backlash	\\	092

Example

Some character constants expressed in terms of escape sequences are given below:

```
'\n'        '\t'        '\b'        '\''        '\\'        '\"'
```

Note that the character constants `'\0'` and `'0'` are distinct. Also, note that the escape sequences can be expressed by means of octal or hexadecimal number systems. Generally, using an octal or hexadecimal escape sequence is less popular than writing the character constant directly.

String constants

A string constant consists of any number of consecutive characters (including none) enclosed in double quotation marks, *e.g.*, "`Operation Flood`"; "`NDRI, Karnal-132001`"; "`+91-184-2559015`"; "`19.95`"; "`The correct answer is: `"; "`2*(I+3)/J`"; "`Line 1 \nLine 2\nLine 3`"; " "; " "; and so on.

Note that a character constant, *e.g.*, `'A'` and corresponding single-character string constant, `"A"` are not equivalent! Also, mind that a character constant has an equivalent integer value, whereas a single-character string constant does not have such an integer value.

4.6 Variables and Arrays

A variable is a named memory location (in the main memory or RAM) that can be used to read and write information. You may think of a variable as placeholder for a value. A variable is considered as being equivalent to its assigned value. Thus, if there is a variable,i initialised (or set equal) to 0, then it follows that i+1 will be equal to 1. Hence, a variable is an identifier that denotes some specified type of information within a designated portion of the programme. The variable must be assigned a value at some point in the programme. Subsequently, the value can be accessed in the programme by referring to the variable name. A given variable can be assigned different values at various places within the programme. Thus, the information contained in the variable can change during the execution of the programme. However, the type of the information (*i.e.*, numeric or string, *etc*.) associated with the variable cannot change. The rules governing naming variables in 'C' are as follows:

- The name can contain letters, digits and the underscore
- The first letter must be any valid letter or the underscore
- An underscore as the first letter should be avoided as it may conflict with standard system variables
- The length of name can be unlimited although the first 31 characters must be unique
- Keywords cannot be used as a variable name

- Of course, the variable name should be meaningful to the programming context.

As ‘C’ is, relatively, a low-level programming language, therefore, before a ‘C’ programme can utilise memory to store a variable, it must demand the memory needed to store the values for a variable. This is realised by declaring variables. Declaring variables is the statement(s) in which a ‘C’ programme specifies the number of variables it needs, their names and quantum of memory they will need.

The array is another kind of variable that is commonly used in ‘C’. An array is an identifier that refers to a collection of data items that all have the same name. The data items must all be of the same type such as integer, character, *etc.* The individual data items are represented by their corresponding array elements, *i.e.*, the first data item is represented by the first array element and so on. The individual array elements are distinguished from each other by the value that is assigned to a subscript. The detailed description of variables and arrays will be made in the following modules/lessons.

4.7 Operators

An operator is a symbol that facilitates the programmer to instruct the computer machine to perform certain mathematical or logical manipulations. Operators are used in ‘C’ programme to operate on data and variables. ‘C’ supports a rich collection of operators, *viz.*, arithmetic operators, relational operators, logical operators, assignment operators, increments and decrement operators, conditional operators, bitwise operators and special operators (to be discussed later).

Lesson 5

DATA TYPES IN 'C'

5.1 Introduction

'C' uses the concept of data types, which is used to define a variable before its use in a programme. The definition of a variable will assign storage for the variable and define the type of data that will be held in the memory location. Hence, the data type defines: the amount of storage allocated to variables; the values that the variables can accept; and the operations that can be performed on these variables. 'C' data types can be broadly classified as:

- Primary data type
- Derived data type
- User-defined data type

5.2 Primary Data Types

All 'C' compilers accept the following basic data types:

Table 5.1 Basic data types supported by 'C'

Serial Number	*Data Type*	*Keyword*	*Range of Values*
1.	Integer	`int`	- 32768 to +32767
2.	Character	`char`	- 128 to 127
3.	Floating-point	`float`	3.4e - 38 to 3.4e + 38
4.	Double precision floating-point	`double`	1.7e - 308 to 1.7e + 308
5.	Void	`void`	-

Integer types

Integers are whole numbers with a machine dependent range of values. 'C' has three classes of integer storage, namely, `short int`, `int` and `long int`. All of these data types have signed and unsigned forms. The `short int` requires half the space than normal integer values. Unsigned numbers are always positive and consume all the bits for the magnitude of the number. The `long` and `unsigned` integers are used to declare a longer range of values.

Floating-point types

A floating-point number represents a real number with six digits precision. Floating-point numbers are denoted by the keyword float. When the accuracy of the floating-point number is insufficient, the data type, double is used to define the number. The double is same as float but with longer precision. To extend the precision further, use long double,which consumes 80 bits of memory space.

Void type

The `void` data type is used to specify the type of a function. It is a good practice to avoid functions that does not return any values to the calling function.

Character type

A single character can be defined as a character type of data. Characters are usually stored in 8 bits of internal storage. The qualifier signed or unsigned can be explicitly applied to `char`. While unsigned characters have value between 0 and 255 characters have values from -128 to 127. Size and range of data types on 16 bit machine is given in Table-5.2 below:

Table 5.2 Size and range of data types on 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	- 32768 to 32767
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	- 2147483648 to 2147483647
unsigned long int	32	0 to 4294967295
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 3.4e + 4932

5.3 Declaration of Variables

Every variable used in the programme should be declared to the compiler. The declaration serves two purposes:

a) Tells the compiler the variables' names.
b) Specifies types of data the variables will hold.

The general format of any declaration is:

```
datatype variable_1, variable_2, …, variable_n;
```

where `variable_1`, `variable_2`, *etc.*, are variable names. Variables are separated by commas. A declaration statement must end with a semicolon.

Examples

```
int sum;
int number, salary;
double average, mean;
```

5.4 Programmer-defined Type Declaration

In 'C', the '`typedef`' feature allows programmers to define new data type(s) that is/are equivalent to existing data type(s) [see Table-5.3]. Once a programmer-defined data type is defined, new identifiers such as variables, arrays, *etc.*, can be declared in terms of the newly defined data type. The general syntax is:

```
typedef type user-defined-type;
```

where '`type`' represents existing data type (including a standard data type or earlier programmer-defined data type) and '`user-defined-type`' refers to the new programmer-defined name given to the data type.

Examples

```
typedef int age;
typedef float marks;
```

Here, '`age`' is a programmer-defined data type, which is equivalent to integer data type. Hence, the variable declaration

```
age male, female;
```

is equivalent to writing

```
int male, female;
```

That is, the variables, '`male`' and '`female`' are regarded as variables of type '`age`', though these are actually variables of integer type.

Table 5.3 The standard data types in 'C'

Data Type	*Description*	*Typical Memory Requirements*
`int`	Integer quantity	2 bytes or 1 word (varies from one computer to another)
`Short`	Short integer quantity (may contain fewer digits that `int`)	2 bytes or 1 word (varies from one computer to another)
`long`	Long integer quantity (may contain more digits that `int`)	1 or 2 words (varies from one computer to another)

`unsigned`	Unsigned (positive) integer quantity (maximum permissible quantity is approximately twice as large as `int`)	2 bytes or 1 word (varies from one computer to another)
`char`	Single character	1 byte
`signed char`	Single character, with numerical values ranging from -128 to 127	1 byte
`unsigned char`	Single character, with numerical values ranging from 0 to 255	1 byte
`float`	Floating-point number (*i.e.*, a number containing a decimal point and/or an exponent)	1 word
`double`	Double-precision floating-point number (*i.e.*, more significant figures, and an exponent that may be larger in magnitude)	2 words
`long double`	Double-precision floating-point number (may be higher precision than double)	2 or more words (varies from one computer to another)
`void`	Special data type for functions that do not return any value	(not applicable)
`enum`	Enumeration constant (special type of `int`)	2 bytes or 1 word (varies from one computer to another)

Note: The qualifier `unsigned` may appear with `short int` or `long int`, *e.g.*, `unsigned short int` (or `unsigned short`), or `unsigned long int` (or `unsigned long`).

Similarly, the following declarations:

```
typedef float height[30];
height boys, girls;
```

define '`height`' as a 30-element floating-point array type. Hence, `boys and girls` are 30-element floating-point arrays (arrays are discussed later). The `typedef` feature is quite suitable while defining structures as it avoids the need to repeatedly write the `struct tag` whenever a structure is referenced. Further discussion will be continued on this topic in the last module.

5.5 Symbolic Constants

Symbolic constants are names for a sequence of characters. The characters may represent a numeric constant, a character constant or a string constant. Thus, a symbolic constant allows a name to appear in place of a numeric, character or string constant. At the time of compiling each occurrence of a symbolic constant gets replaced by its corresponding character sequence. These are usually defined at the beginning of a programme, *e.g.*, the following statements:

```
#define ANGLE_MIN 0
#define ANGLE_MAX 360
```

```
#define PI 3.141593
#define TRUE 1
#define FALSE 0
```

would define the symbolic constants, `ANGLE_MIN`, `ANGLE_MAX`, `PI`, `TRUE` and `FALSE` to the values `0`, `360`, `3.141593`, `1` and `0`, respectively. Recall that 'C' distinguishes between lowercase and uppercase letters in variable names. It is a tradition to use capital letters in defining global constants (like symbolic constants). Note that the symbolic constant definitions do not end with a semicolon unlike other 'C' statements.

Important Note: The following Sections 5.6 and 5.7 deal with the advanced topics on Enumerations and Macros. Therefore, the students may better comprehend these advanced topics after completing other lessons of this module.

5.6 Enumerations

An enumeration is a data type like a `structure` or a `union` (to be discussed later). It consists of a set of named values that represent integral constants, known as enumeration constants. An enumeration is also referred to as an enumerated type because you must list (enumerate) every value in creating a name for each of them. In addition to providing a way of defining and grouping sets of integral constants, enumerations are useful for variables that have a small number of possible values. You can declare an enumeration type separately from the definition of variables.

Enumeration type definition

Generally, an enumeration type definition begins with the `enum` keyword followed by an optional identifier (the enumeration tag) and a brace-enclosed list of enumerators. A comma separates each enumerator in the enumerator list.

```
enum tag {member 1, member 2, …, member m};
```

where `enum` is required keyword; *`tag`* is a name that identifies enumerations having this composition; and *`member 1`*, *`member 2`*, …, *`member m`* represent the individual identifiers that may be assigned to variables of this type. These member names must unique as well as they must be distinct from other identifiers whose scope is the same as that of the enumeration.

Enumeration variable declaration

Once the enumeration is defined, corresponding enumeration variables can be declared as follows:

```
storage-class enum tag variable 1, variable 2, …, variable n;
```

where *`storage-class`* is an optional storage class specifier, `enum` is the required keyword, *`tag`* is the name that appeared in the enumeration definition, and *`variable 1`*, *`variable 2`*, …, *`variable n`* are enumeration variables of the type *`tag`*.

The enumeration definition can be clubbed with the variable declarations, as follows:

```
storage-class enum tag {member 1, member 2, …, member m}
        variable 1, variable 2, …, variable n;
```

the *tag* is optional in this situation.

An illustration – Consider the following statements as a part of a 'C' programme:

```
enum colours {black, blue, cyan, green, magenta, red, white,
yellow};
colours foreground, background;
```

Note that, the first statement defines enumeration named `colours` (*i.e.*, the *tag* is *colour*). The enumeration consists of eight constants whose names are `black, blue, cyan, green, magenta, red, white` and `yellow`. The second statement declares the variables foreground and background to be enumeration variables of type colours. Thus, each variable can be assigned any one of the constants `black, blue, …, yellow`

The two declarations can be combined as follows:

```
enum colours {black, blue, cyan, green, magenta, red,white,
yellow} foreground background;
```

or without the tag, simply:

```
enum {black, blue, cyan, green, magenta, red, white,yellow}
foreground background;
```

Enumeration constants are automatically assigned equivalent integer values, beginning with 0 for the first constant and with each successive constant increasing by 1. Thus, *member 1* will automatically be assigned the value `0`; *member 2* will be assigned `1`, and so on.

Example 1: Consider the following code demonstrating the use of enumeration constants. (For detailed description about the `printf` statement, see Lesson-5).

```
#include <stdio.h>
int main ()
{
  enum compass_direction  {north, east, south, west};
  enum compass_direction my_direction;
  my_direction = east;
  printf("%d", my_direction);
  return 0;
}
```

Output :

It is quite interesting to note that the aforementioned automatic assignments can be overridden within the definition of the enumeration! That is, some of the constants can be assigned explicit integer values, which differ from default values. To do so, each constant (*i.e.*, each *member*), which is assigned an explicit value is expressed as an ordinary assignment expression; *member = int*, where *int* represents a signed integer quantity. Those constants that are not assigned

explicit values will automatically be assigned values, which increase successively by 1 from last explicit assignment. This may cause two or more enumeration constants to have the same integer value.

Example 2: Consider the following code demonstrating the use of enumeration constants with implicit and explicit assignments.

```
#include <stdio.h>
int main ()
{
  enum colour  {black=-1, blue, cyan, green,
         magenta, red=2, white, yellow};
  enum foreground background;
  background = green;
  printf("%d\n", background);
  return 0;
}
```

Output :

The constants black and red are now assigned the explicit values,-1 and 2 respectively. The remaining enumeration constants are automatically assigned values that increase successively by 1 from the last explicit assignment. Thus, blue, cyan, green and magenta are assigned the values. 0, 1, 2 and 3 respectively. Similarly, white and yellow are assigned the values 3 and 4, respectively. Note that there are now duplicate assignments, i.e., green and red represents 2, where magenta and white both represent 3.

Enumeration variables can be processed in the same manner as other integer variables. Thus, they can be assigned new values, compared, *etc*. However, it should be understood that enumeration variables are generally used internally, to indicate various conditions that can arise within a programme. Hence, there are certain restrictions associated with their use. In particular, an enumeration constant cannot be read into the computer and assigned to an enumeration variable. (It is possible to enter an integer and assign it to an enumeration variable, though it is generally not done). Moreover, only integer value of an enumeration variable can be written out of the computer.

5.7 Macros

As discussed earlier, the `#define` statement is used to define symbolic constants within a 'C' programme. All symbolic constants are replaced by their equivalent text at the beginning of the compilation process. Thus, symbolic constants provide shorthand notation to simplify the organisation of a programme. Besides, `#define` statement can be used to define macros. A macro is a single identifier that is equivalent to expressions, complete statements or group of statements; *i.e.*, a fragment of code, which has been given a name. Whenever this name is used

within the programme, it is replaced by the contents of the macro. In this sense, macros resemble functions; however, they are defined in an entirely different manner than functions.

You may define any valid identifier as a macro, even if it is a 'C' keyword. The pre-processor does not know about the keywords. This can be useful if you wish to hide a keyword such as `const` from an older compiler that does not recognise it. However, the pre-processor operator '`defined`' can never be defined as a macro. Macros slow down the compiling process; however, the compiled programmes (executable codes) are faster than functions as functions involve passing values thereby increasing CPU usage.

The formal syntax of a macro is:

```
#define name(dummy1[,dummy2][,...]) token string
```

The symbols `dummy1`, `dummy2`, `...` are called dummy arguments (the square brackets indicate optional items).

Example 1:

Consider the following simple example of a macro *(Students should never emulate this in any real project)*:

```
#define SquareOf(x) x*x
```

It defines a kind of function, which, used in an actual piece of code, looks exactly like any other function call:

```
double y_out, x_in=3;
y_out = SquareOf(x_in);
```

As you would see subsequently, the problem is that the macro '`SquareOf`' only pretends to be a function call, while it is absolutely different.

There are a few additional rules such as that the macro can extend over several lines, provided one uses a backslash to indicate line continuation:

```
#define ThirdPowerOf(dummy_argument) \
         dummy_argument \
        *dummy_argument \
        *dummy_argument
```

Of course, you should break the line at a reasonable position; and not, for example, in the middle of a symbol.

How does a compiler handle a macro?

What makes a macro different from a standard function is primarily the fact that a macro is a scripted directive for the compiler rather than a scripted piece of run-time code; and, therefore, it is dealt with at compilation time rather than at run time. When the compiler encounters a previously defined macro, it first isolates its actual arguments, handling them as plain text strings

separated by commas. Then it parses (*i.e.*, divides the code into functional components; compiler must parse source code in order to translate it into object code) the *token string*, isolates all occurrences of each dummy-argument symbol and replaces it by the actual argument string. The whole process consists entirely of mechanical string substitutions with almost no semantic (logical) testing!

The compiler then substitutes the modified *token string* for the original macro call and compiles the resulting code script. It is only in that phase that compilation errors can occur. When they do, the result is often either amusing or frustrating, depending upon how you feel at that moment as you may get mysteriously looking error messages resulting from the modified text; and thus, referring to something you have never written!

This is explained with the help of the following small programme, which is formally correct and compiles without any problem:

```
#include < stdio.h>
#define SquareOf(x) x*x
void main()
{
   int x_in=3;
   printf("\nx_in=%i",x_in);
    printf("\nSquareOf(x_in)=%i",SquareOf(x_in));
    printf("\nSquareOf(x_in+4)=%i",SquareOf(x_in+4));
    printf("\nSquareOf(x_in+x_in)=%i",SquareOf(x_in+x_in));
}
```

Naturally, you would expect the output of this programme as:

```
x_in=3
SquareOf(x_in)=9
SquareOf(x_in+4)=49
SquareOf(x_in+x_in)=36
```

However, what you actually get is:

```
x_in=3
SquareOf(x_in)=9
SquareOf(x_in+4)=19
SquareOf(x_in+x_in)=15
```

Let us see what happened. When the complier encountered the string “Squareof(x_in+4)”, it replaced the string with the string “x*x”; followed by replacing each of the dummy-argument-string “x_in+4”, obtaining the final string “x_in+4 * x_in+4”, which, in fact, evaluates to 19 and not to the expected value of 49. Similarly, it is now easy to work out the expression `SquareOf (x_in+x_in)` and understand why and how the result differs from the expected one.

The problem would have never happened if `SquareOf(x)` were a normal function. In that case, the argument `x_in+4` would be first evaluated as a self-standing expression and only then would the result be passed to the function `SquareOf` for the evaluation of the square.

Actually, both the ways are correct! They are just two different recipes on how to handle the respective scripts. However, given the formal similarity between the function-like macro call and a standard function call; the discrepancy is dangerous and should be removed. Luckily, there is a simple remedy, *i.e.*, replace the original definition of the `SquareOf` macro by

```
#define SquareOf(x)  (x)*(x)
```

The problem vanishes because, for example, the macro-call string "`SquareOf(x_in+4)`" is transformed into "`(x)*(x)`" and then into "`(x_in+4)*(x_in+4)`", which evaluates exactly as intended.

On the basis of the foregoing discussion about macros, the students are advised to keep in mind the following rules while using macros in their programmes (For detailed description about the `do...while` statement, see Module-III):

Rule 1: Always write multi-line macros using following pattern:

```
#define name \
   do { \
      macro definition here  \
   } while (0)
```

Rule 2: Always surround macro arguments with parentheses inside the macro body.

Rule 3: Keep your macros as short as possible.

Example 2:

```
#include < stdio.h>
#define MUL(x,y) (x) * (y)
int main()
{
    printf("%d\n" , MUL(3, 5));
//    system("pause");
    return 0;
}
```

Example 3:

```
/*
 * It swaps two integer numbers.
 * Requires tmp variable to be defined.
 */
#define SWAP(x, y) \
```

```
    do { \
      tmp = x; \
      x = y; \
      y = tmp; } \
    while (0)
int main()
{
     int x=10, y=20;
     SWAP(x,y);
     printf("%d %d\n" , x, y);
     return 0;
}
```

Lesson 6

INPUT/OUTPUT STATEMENTS IN 'C'

6.1 Data Input and Output

Reading data and writing the processed data, *i.e.*, information, are two essential operations performed by a typical 'C' programme. Generally, the input data are read by the machine either by means of a standard input device like keyboard or a programmer-defined data file (which is usually a text file that may be created using notepad like text-editors). Similarly, the information is displayed via a standard output device such as monitor, and/or written onto a printer or saved as a user-defined output file. The method for assigning data values to the variables you have seen so far is the assignment statement. However, 'C' supports several functions, *viz.*, `getchar`, `putchar`, `scanf`, `printf`, `gets` and `puts` for transferring the data or information between computer and standard input/output (I/O) devices. The first two functions, `getchar` and `putchar` facilitate only single characters to be transferred into and out of computer; the `scanf` and `printf` functions allow the transfer of single characters, numerical values and strings; and the functions, `gets` and `puts` support the input and output of strings.

The I/O functions can be accessed from anywhere within a programme merely by writing the function name followed by a list of arguments enclosed in parentheses. The arguments correspond to data items being sent to the function. Some I/O functions do not involve arguments but the empty parentheses must appear even in such cases. Most versions of 'C' include a collection of header files that provides necessary information in support of the various library functions.

6.2 Header Files

'C' uses header files that allow programmers to split certain elements of a source programme into reusable files. Generally, the header files contain forward declarations of subroutines, variables and other identifiers. If programmers want to declare standardised identifiers in more than one source file, they can place such identifiers in a single header file, which can be included in other codes whenever the header contents are required. This is to keep the interface in the header separate from the implementation. Hence, a header file is a file containing 'C' declarations and macro definitions to be shared between several source files. You may request the use of a header file in your 'C' programme by including it, with the 'C' pre-processing directive, '`#include`', *e.g.*, the header file required by the standard I/O library functions is `stdio.h`. The 'C' statement for this purpose is written as

```
#include <stdio.h>.
```

'C' pre-processor

The C pre-processor is a separate step in the compilation process, but it is not a part of the compiler. 'C' pre-processor is just a text substitution tool. All pre-processor statements always begin with the number sign (or hash symbol), #. The unconditional directives are as follows:

- `#define` – Defines a pre-processor macro *(discussed later in this lesson)*
- `#include` – Inserts a particular header from another file
- `#undef` – Un-defines a pre-processor macro

Example 1

The following statement instructs the 'C' pre-processor to replace instances of `MAX_ARRAY_LENGTH` with 20:

```
#define MAX_ARRAY_LENGTH 20
```

You should use `#define` for defining constants in 'C' programmes so as to increase readability of the code.

Example 2

Similarly, the following statement directs the 'C' pre-processor to get `mystring.h` header file from the local directory/folder and then add the text to the file:

```
#include "mystring.h"
```

Example 3

The following code segment guides the 'C' pre-processor to un-define `MEANING_OF_LIFE` and define it for the constant 42:

```
#undef MEANING_OF_LIFE
#define MEANING_OF_LIFE 42
```

Refer the books given under 'Suggested Readings' for the conditional and other types of pre-processor directive statements.

Now, let us resume the discussion about the header files. Header files serve two purposes:

- System header files declare the interfaces to parts of the operating system. You may include these header files in your 'C' programme to supply the definitions and declarations you need to invoke system calls and libraries.
- User defined header files contain declarations for interfaces between the source files of the user's programme. Each time the user has a group of related declarations and macro definitions all or most of which are needed in several different source files, it is a good idea to create a header file for them.

Although, including a header file produces the same results as copying the contents of the header file into each source file that needs it, but such copying would be time-consuming and error-prone. However, a header file facilitates the related declarations to appear only at a single place. In case of any change in the content of the header file, the change is made in one place and programmes that include the header file will automatically use its (header file's) latest version when the programme is compiled afresh. Thus, the header file eliminates the labour of finding and changing all the copies as well as the risk that a failure to find one copy will result in inconsistencies within a programme.

In 'C', the usual convention is to give header files names that end with '`.h`'. Generally, only letters, digits, dashes, underscores and at most one dot are used in header file names. Every external function is mentioned in a header file, including libraries that are pre-compiled into object code, and source files required to build the 'C' programme.

6.3 Standard Header Files

There are many operations that are not catered for in the 'C' programming language, for instance, string manipulation. However, all compilers of 'C' provide a set of libraries to provide this missing functionality, which are collectively known as the standard libraries. A list of header files is given in Table-6.1.

Table 6.1 Standard 'C' header files

Name	*Description*
`assert.h`	Contains the assert macro, used to assist with detecting logical errors and other types of bug in debugging versions of a programme.
`complex.h`	A set of functions for manipulating complex numbers.
`conio.h`	Console I/O Routines used in old MS-DOS compilers to create text user interfaces *(not a part of standard library)*.
`ctype.h`	Contains functions used to classify characters by their types or to convert between upper and lower case in a way that is independent of the used character set.
`errno.h`	For testing error codes reported by library functions.
`fenv.h`	For controlling floating-point environment.
`float.h`	Contains defined constants specifying the implementation-specific properties of the floating-point library, such as the minimum difference between two different floating-point numbers (`_EPSILON`), the maximum number of digits of accuracy (`_DIG`) and the range of numbers, which can be represented (`_MIN`, `_MAX`).
`inttypes.h`	For precise conversion between integer types.
`iso646.h`	For programming in ISO 646 variant character sets.
`limits.h`	Contains defined constants specifying the implementation-specific properties of the integer types, such as the range of numbers, which can be represented (`_MIN`, `_MAX`).
`locale.h`	For `setlocale` and related constants. This is used to choose an appropriate locale.
`math.h`	For computing common mathematical functions.
`setjmp.h`	Declares the macros `setjmp` and `longjmp`, which are used for non-local exits.
`stdarg.h`	For accessing a varying number of arguments passed to functions.
`signal.h`	For controlling various exceptional conditions.
`stdbool.h`	For a Boolean data type.

stdint.h	For defining various integer types.
stddef.h	For defining several useful types and macros.
stdio.h	Provides the core input and output capabilities of the 'C' language. This file includes the venerable printf function.
stdlib.h	For performing a variety of operations, including conversion, pseudo-random numbers, memory allocation, process control, environment, signalling, searching and sorting.
string.h	For manipulating several kinds of string.
tgmath.h	For type-generic mathematical functions.
time.h	For converting between various time and date formats.
wchar.h	For manipulating wide streams and several kinds of string using wide characters - key to supporting a range of languages.
wctype.h	For classifying wide characters.

6.4 Single Character Input – The `getchar` Function

Single character values can be input to a programme variable (computer memory) using 'C' library function, `getchar`. This function belongs to the standard I/O Library supported by 'C'. It returns a single character from a standard input device (keyboard). The function does not require any arguments, though a pair of parentheses must follow the keyword, `getchar`. In general form, a function reference is written as:

```
character variable = getchar();
```

where character variable refers to some previously declared character variable. For instance, consider the following 'C' code segment:

```
char c;
. . .
c = getchar();
```

The first statement declares that `c` is a character type variable. The second statement causes a single character to be input from the standard input device (usually a keyboard) and then assigned to the variable, `c`. Note that `getchar` function can also be used to read multi-character strings by reading one character at a time using a suitable loop (see Module III for detailed description about loop control statements).

6.5 Single Character Output – The `putchar` Function

Single characters can be displayed using 'C' library function `putchar`. This function is complementary to the character input function `getchar`. This function is also a part of the standard I/O Library supported by 'C'. It transmits a single character to a standard output device (usually a monitor). The character being transmitted is represented as a character type variable. It

must be expressed as an argument to the function enclosed in parentheses following the keyword, `putchar`.

For instance, consider the following 'C' code segment:

```
char c;
. . .
putchar(c);
```

The first statement declares that `c` is a character type variable. The second statement causes the current value of `c` to be transmitted to the standard output device (*e.g.*, monitor) where it will be displayed. Note that `getchar` function can also be used to output a string constant by storing the string within a one-dimensional character type array. Each character can then be written separately within a loop (discussed in Module-III).

6.6 Entering Input Data – The `scanf` Function

Input data can be entered into the computer's memory from a standard input device by means of the 'C' library function `scanf`. This function facilitates to enter any combination of numerical values, single characters and strings. The function returns the number of data items that have been entered successfully. The general syntax of the `scanf` function is:

`scanf(`*`control string`*`, `*`arg_1`*`, `*`arg_2`*`, …, `*`arg_n`*`)`

where *`control string`* refers to a string containing certain required formatting information and `arg_1`, `arg_2`, …, `arg_n` are arguments that represent the individual input data items. In fact, the arguments correspond to pointers that lead to the addresses of the data items within the computer's memory.

The control string consists of individual groups of characters, with one character group for each input data item. Each character group must begin with a percent symbol (`%`). In its simplest form, a single character group will consist of the percent symbol followed by a conversion character, which indicates the type of the corresponding data item.

Within the control string, multiple character groups can be contiguous, or they can be separated by white space characters, *i.e.*, blank spaces, tabs or newline characters. If white space characters are used to separate multiple character groups in the control string, then all consecutive white space characters in the input data will be read but ignored. The use of blank spaces as character-group separators is a very common practice among the 'C' programmers. The commonly used conversion characters are listed in Table-6.2.

The arguments are written as variables or arrays, whose types match the corresponding character groups in the control string. Each variable name must be preceded by an ampersand (`&`). The arguments are actually pointers that indicate where the data items are stored in computer's memory. However, array names should not begin with an ampersand.

Table 6.2 Commonly used `scanf` Conversion Characters

Conversion Character	*Description*
c	Data item is a single character
d	Data item is a decimal integer
e	Data item is a floating-point value
f	Data item is a floating-point value
g	Data item is a floating-point value
h	Data item is a short integer
i	Data item is a decimal, hexadecimal, or octal integer
o	Data item is an octal integer
s	Data item is a string followed by a white space character (the null character '\0' will automatically be added at the end)
u	Data item is an unsigned decimal integer
x	Data item is a hexadecimal integer
[. . .]	Data item is a string, which may include white space characters

A *prefix* may precede certain conversion characters.

Prefix	*Description*
h	Short data item (short integer or short unsigned integer)
l	Long data item (long integer, long unsigned integer or double)
L	Long data item (long double)

Example 4: The following segment of a 'C' programme depicts a typical application of the scanf function.

```
char part_name[20];
int part_no;
float unit_price;
. . .
scanf("%s %d %f", part_name, &part_no, &unit_price);
```

ove example, the control string within the scanf function is "%s %d %f". it contains three character groups. The first character group, %s indicates that the ient (part_name) represents a string; the second character group, %d indicates cond argument (&part_no) represents a decimal integer; and the third character : indicates that the third argument (&unit_price) represents a floating-point e that the numerical variables part_no and unit_price are prefixed by an

within the scanf function. However, an ampersand does not precede me being an array name. Also, note that the aforementioned scanf statement can as

```
scanf("%s%d%f", part_name, &part_no, &unit_price);
```

y white space characters in the control string. This is valid.

data items entered through a standard input device (keyboard) are numeric values, acters or strings or some combination thereof. These data items must correspond in type and in order to the arguments mentioned in the scanf function. Numeric supplied same as numeric constants while floating-point values must have either a oint or an exponent (or both). If two or more data items are entered, they must be by white space characters. The data items may run in two or more lines, because e character is considered to be a white space character and can, therefore, separate e data items. Moreover, if the control string begins with reading a character type it is generally a good idea to precede the first conversion character with a blank s causes the scanf function to ignore any extraneous characters that may have ed earlier, *e.g.*, the following segment of a 'C' programme depicts this:

```
char part_name[20];
int part_no;
float unit_price;
. . .
scanf(" %s %d %f", part_name, &part_no, &unit_price);
```

lank space that precedes the %s. This prevents any previously entered superfluous from being assigned to part_name. The following data values can be entered ard input device while the programme is executed.

```
button 12345 0.002
```

characters that make up the string button would be assigned to the first six f the array variable, part_name; the integer value, 12345 would be assigned to le part_no, and the floating-point value 0.002 would be assigned to the unit_price. Note that individual items are entered on one line separated with es. However, these data values could be entered on separate lines too as newline are considered as white space characters. Hence, the data values could be entered ne following three ways:

(i)
```
Button
12345
0.002
```

(ii)
```
Button
12345 0.002
```

(iii)
```
button 12345
0.002
```

Note that the s-type conversion character applies to a string that is terminated by a white space character. Thus, the string containing white space characters cannot be entered in this manner rather there are ways to deal with such strings that include white space characters. One way is to use the `getchar` function within loop (see Module III). Alternatively, you can use `scanf` function to enter such strings. For this, the s-type conversion character within the control string is replaced by a sequence of characters enclosed in square braces. White space characters may be incorporated within the brackets so as to accommodate strings with such characters. When the programme is running, successive characters will continue to be read from the standard input device as long as each input character matches one of the characters enclosed within the square braces. The order of characters within the square braces need not correspond to the order of characters being entered. Input characters can be repeated. The string will terminate once an input character is encountered that does not match any of the characters specified within the square braces. A null character (`\0`) will then automatically be added to the end of the string.

Example 5: The following programme segment demonstrates the use of `scanf` function to feed a string (of undetermined length but restricted to maximum 80 characters including the null character) consisting of uppercase letters and blank spaces. Note the blank space that precedes the `%` symbol in the `scanf` statement.

```
char line[80];
. . .
scanf(" %[ABCDEFGHIJKLMNOPQRSTUVWXYZ]", line);
```

If the input string

```
NEW DELHI CITY
```

is entered from the standard input device when the programme is executed, the entire string will be assigned to the array, `line` as the string is comprised of uppercase letters and blank spaces. However, if the string were written as

```
New Delhi City
```

then only the single letter, `N` would be assigned to the array variable, `line`, because the first lowercase letter, *i.e.*, 'e' in this example, would be interpreted as the first character beyond the string.

6.7 Writing Output Data – The `printf` Function

The information generated by a programme after processing the input data can be transmitted from computer to a standard output device (typically a monitor or printer) using the library function `printf`. This function can be used to output any combination of numerical values, single characters and strings. The general syntax for the `printf` function is

```
printf(control string, arg_1, arg_2, …, arg_n);
```

where control string refers to a string that contains formatting information and `arg_1`, `arg_2`, …, `arg_n` are arguments that represent the individual output data items. The arguments can be written as constants, single variable or array variable or even more complex expressions; function references may also be included. Mind it carefully that unlike `scanf` function, the arguments in `printf` function do not represent memory addresses and, therefore, are not preceded by ampersand. The control string consists of individual control groups of characters, with one character group for each output data item. Each control group must begin with a percent symbol (`%`) followed by a conversion character (as given in Table-6.3) specifying the type of the corresponding data item. Multiple character groups can be contiguous or they can be separated by other characters including white space characters. These 'other' characters are transferred directly to the output device(s), where they are displayed and/or written. The use of blank space(s) as character group delimiters is a common practice. A list of commonly used conversion characters is given in Table-6.3.

Table 6.3 `printf` conversion characters

Conversion Character	*Description*
`c`	Data item is displayed as a single character
`d`	Data item is displayed as a signed decimal integer
`e`	Data item is displayed as a floating-point value with an exponent
`f`	Data item is displayed as a floating-point value without an exponent
`g`	Data item is displayed as a floating-point value using either `e`-type or `f`-type conversion, depending on value; trailing zeros, trailing decimal point will not be displayed.
`i`	Data item is displayed as a signed decimal integer
`o`	Data item is displayed as an octal integer, without a leading zero
`s`	Data item is displayed as a string
`u`	Data item is displayed as an unsigned decimal integer
`x`	Data item is displayed as a hexadecimal integer, without leading `0x`

Some of these characters are interpreted differently than with the `scanf` function.

A *prefix* may precede certain conversion characters.

Prefix	*Description*
`h`	Short data item (short integer or short unsigned integer)
`l`	Long data item (long integer, long unsigned integer or double)

L	Long data item (long double)

Example 6:

```
#include <stdio.h>
#include <math.h>
. . .
float i = 2.0, j = 3.0;
printf("%f %f %f %f", i, j, i+j, sqrt(i+j));
. . .
```

Note that the first two arguments within the `printf` function are single variables; the third argument is an arithmetic expression and the last argument is a function reference that has a numeric expression as an argument. Executing the programme (after completing the missing statements to be discussed later) would produce the following output;

```
2.000000        3.000000        5.000000        2.236068
```

Example 7:

The following programme segment demonstrates as to how different types of data can be displayed using the `printf` function:

```
char part_name[20];
int part_no;
float unit_price;
. . .
scanf("%s %d %f", part_name, &part_no, &unit_price);
. . .
printf("%s %d %f", part_name, part_no, unit_price);
```

Example 8:

The following 'C' programme converts the given length in feet to centimetre. You know that one foot contains 30.48 cm. (See Example-1 of Lesson-2 for pseudo code for this programme)

Note: *Students may like to refer to the Section 7.7 of Lesson-7 for structure of a typical 'C' programme.*

```
#include <stdio.h>
void main ()
{
 float len_ft, len_cm;
 printf("Length in feet: ");
 scanf("%f",& len_ft);
 len_cm=len_ft*30.48;
```

```
        printf("\nLength in feet is = %f", len_ft);
        printf("\nLength in cm is = %f", len_cm);
        getch();
       }
```

Input

```
Length in feet : 1
```

Output

```
Length in feet is = 1

Length in cm is = 30.48
```

Lesson 7

OPERATORS AND EXPRESSIONS – PART – I
(Arithmetic, Assignment and Relational Operators)

7.1 Arithmetic Operators

All the basic arithmetic operations such as addition, multiplication, division, subtraction and remainder of integer division calculation can be carried out in 'C'. The arithmetic operators are given in Table -7.1.

Table 7.1 'C' Arithmetic operators

Serial Number	*Operator*	*Description*
1.	+	Addition or Unary Plus
2.	-	Subtraction or Unary Minus
3.	*	Multiplication
4.	/	Division
5.	%	Modulus Operator (Remainder after integer division)

Note that there is no exponentiation operator in 'C'. However, 'C' provides an inbuilt function, `pow` (see Table-14.1 for details) for the purpose. The operands connected through arithmetic operators must be numeric values, *i.e.*, integer, floating-point or characters (recall that the character constants represent integer values as determined by the machine's character set). The modulus operator expects both the operands to be integers as well as the second operand to be nonzero. Similarly, the division operator requires the second operand to be non-zero. Evidently, both unary as well as binary operators are supported by 'C'. Unary operation is performed on a single operand, *e.g.*, the unary minus (-) operator applied on the number 5 will produce the resultant value as -5. The operators follow certain precedence rules similarly as the rules of algebra in mathematics. For example, the following expressions demonstrate the use of various arithmetic operators:

$a * b + c$ $\quad -a * b$ $\quad x + y$ $\quad x - y$ $\quad -x + y$

Here a, b, c, x and y are known as operands. The modulus operator is a special operator in 'C', which evaluates the remainder of the operands after division.

7.2 Integer Arithmetic

When an arithmetic operation is performed on two whole numbers or integers then such an operation is called integer arithmetic. It always produces an integer (*i.e.*, truncating the decimal portion of the quotient) as a result.

Illustration

Consider x = 27 and y = 5 be two integer numbers. Thus, the results of various arithmetic operations performed on these numbers are shown below.

Expression	Value	Expression	Value	Expression	Value
$x + y$	32	$x - y$	22	$x * y$	135
$x \% y$	2	x / y	5		

Note that the fractional part is truncated during integer division.

7.3 Floating-point Arithmetic

The process of applying an arithmetic operation on two real numbers is called floating-point arithmetic. The floating-point results can be truncated according to the properties requirement. Note that the modulus operator is not applicable to floating-point arithmetic operands.

Illustration

Consider x = 14.0 and y = 4.0 be two real numbers. Thus, the results of various arithmetic operations performed on these numbers are shown below.

Expression	Value	Expression	Value	Expression	Value	Expression	Value
$x + y$	18.0	$x - y$	10.0	$x * y$	56.0	x / y	3.50

Note that the interpretation of the remainder operation is ambiguous when one of the operands is negative. Most versions of 'C' assign the sign of the first operand to the remainder. Most versions of 'C' ascertain the sign of the remainder as a = ((a/b) * b) + (a%b); however, this aspect is not mentioned in the formal definition of the language. Thus, the aforesaid condition will always be fulfilled regardless of the signs of the operands a and b. Novice programmers should be careful while using the modulus operator when one of the operands is negative.

Illustration

Consider a and b be two integer variables containing values 11 and -3, respectively. Several arithmetic expressions involving these variables and their resultant values are given below.

Expression	Value	Expression	Value
$a+b$	8	$a-b$	14
$a*b$	-33	a/b	-3
$a\%b$	2		

Operands with distinct data types commonly go through type conversion before the expression attains the final value. Generally, the final result is expressed in the highest precision possible, consistent with the data types of the operands. The following rules apply when neither operand is unsigned.

7.3.1 Conversion rules

These rules apply to arithmetic operations between two operators with dissimilar data types. There may be some variation from one version of 'C' to another.

1. If one of the operands is `long double`, the other will be converted to `long double` and the result will be `long double`.
2. Otherwise, if one of the operands is `double`, the other will be converted to `double` and the result will be `double`.
3. Otherwise, if one of the operands is `float`, the other will be converted to `float` and the result will be `float`.
4. Otherwise, if one of the operands is `unsigned long int`, the other will be converted to `unsigned long int` and the result will be `unsigned long int`.
5. Otherwise, if one of the operands is `long int` and the other is `unsigned int`, then:
    a. If `unsigned int` can be converted to `long int`, the `unsigned int` operand will be converted as such and the result will be `long int`.
    b. Otherwise, both operands will be converted to `unsigned long int` and the result will be `unsigned long int`.
6. Otherwise, if one of the operands is `long int`, the other will be converted to `long int` and the result will be `long int`.
7. Otherwise, if one of the operands is `unsigned int`, the other will be converted to `unsigned int` and the result will be `unsigned int`.
8. If none of the above conditions applies, then both operands will be converted to `int` (if necessary), and the result will be `int`.

Note that some versions of 'C' automatically convert all floating-point operands to double-precision.

Further, note that the 'C' operators are grouped hierarchically according to their precedence (*i.e.*, the order of evaluation) as shown in Table-7.2.

Operations with higher precedence are evaluated prior to the lower precedence operations. However, the normal order of evaluation can be transformed by using parentheses, *e.g.*, the expressions, a – b/c * d and (a – b)/(c * d) will produce different results. At times, it is a good practice to use parentheses to simplify an expression, even if the parentheses may not be required. On the other hand, very complex expressions should be avoided as these may be error-prone.

Table 7.2 'C' operators in order of precedence (highest to lowest) along with their associativity that indicates in what order operators of equal precedence in an expression are applied

Description	Associativity

Operator		Group
`()` `[]` `.` `->` `++ --`	Parentheses (function call) (see Note 1) Brackets (array subscript) Member selection via object name Member selection via pointer Postfix increment/decrement (see Note 2)	Left-to-Right
`++ --`	Prefix increment/decrement	Right-to-Left
`+ -`	Unary plus/minus	
`! ~`	Logical negation/bitwise complement	
`(type)`	Cast (change *type*)	
`*`	Dereference	
`&`	Address	
`sizeof`	Determine size in bytes	
`* / %`	Multiplication/division/modulus	Left-to-Right
`+ -`	Addition/subtraction	Left-to-Right
`<< >>`	Bitwise shift left, Bitwise shift right	Left-to-Right
`< <= > >=`	Relational less than/less than or equal to Relational greater than/greater than or equal to	Left-to-Right
`== !=`	Relational is equal to/is not equal to	Left-to-Right
`&`	Bitwise AND	Left-to-Right
`^`	Bitwise exclusive OR	Left-to-Right
`|`	Bitwise inclusive OR	Left-to-Right
`&&`	Logical AND	Left-to-Right
`||`	Logical OR	Left-to-Right
`?:`	Ternary conditional	Right-to-Left
`=`	Assignment	Right-to-Left
`+= -=` `*= /=` `%= &=` `^= |=`	Addition/subtraction assignment Multiplication/division assignment	

<<=	Modulus/bitwise AND	
>>=	assignment	
	Bitwise exclusive/inclusive OR	
	assignment	
	Bitwise shift left/right	
	assignment	
,	Comma (separate expressions)	Left-to-Right

Note 1:

Parentheses are also used to group sub-expressions to force a different precedence; such parenthetical expressions can be nested and are evaluated from inner to outer.

Note 2:

Postfix increment/decrement have higher precedence, but the actual increment or decrement of the operand is delayed (to be accomplished sometime before the statement completes execution). So in the statement `y = x * z++;` the current value of `z` is used to evaluate the expression (*i.e.,* `z++` evaluates to `z`) and `z` only incremented after all else is done.

7.4 Mixed-mode Arithmetic

The process of executing an arithmetic expression involving two different kinds of operand, *i.e.*, a real operand and an integer operand is called mixed-mode arithmetic. In case one of the operands is of floating-point type, the result is always of the type, real. For instance, $15/10.0 = 1.5$.

7.5 Assignment Operators

'C' assignment operators are used to assign the value of an expression to an identifier. The syntax of assignment operators is:

identifier = expression

Examples: Following typical assignment expressions depict use of the commonly used assignment operator, '='.

(i) a = 3 (ii) delta = 0.001 (iii) area = length * breadth

The first assignment expression causes the integer value 3 to be assigned to the variable a, and the second assignment causes the floating-point value 0.001 to the variable delta. The third assignment causes the value of the RHS arithmetic expression being assigned to the Left Hand Side (LHS) variable, *i.e.*, the value of the expression length * breadth is assigned to the variable area.

Note: The assignment operator '=' and the equality operator '==' are distinct. Hence, these operators are used for their specific purpose only and are not interchangeable. If the two operands in an assignment expression are of different data types, then the value of the expression on RHS will automatically be converted to the type of the identifier on LHS. In some cases, this automatic

type conversion can cause an alteration of the data being assigned. Some instances for better comprehension of the students are:

- A floating-point value may be truncated if assigned to an integer identifier
- A double-precision value may be rounded if assigned to a single-precision floating-point identifier
- An integer quantity may be changed if assigned to a shorter integer identifier or a character identifier, *i.e.*, some high-order bits are lost
- The value of a character constant assigned to a numeric-type identifier will be dependent upon the particular character set in use, which may cause inconsistencies from one version of 'C' to another version.

Hence, the careless use of type conversions is a frequent source of error among the beginners.

Examples

Consider i as an integer variable.

Assignment Expression	**Resultant Value**
$i = 3.3$	3
$i = 3.9$	3
$i = -3.9$	-3

Now, consider two integer variables, i and j; contains a value of 5. Several assignment expressions that make use of these variables are given below for illustration:

Assignment Expression	**Resultant Value**
$i = j$	5
$i = j/2$	2
$i = 2 * j/2$	5
$i = 2 * (j/2)$	4

Further, assume that i is an integer variable and ASCII character set is applicable.

Assignment Expression	**Resultant Value**
$i = 'x'$	120
$i = '0'$	48
$i = ('x' - '0')/3$	24
$i = ('y' - '0')/3$	24

'C' supports multiple assignments as follows:

$$identifier\ 1 = identifier\ 2 = expression$$

These assignment operations are performed from right to left. Thus, the aforementioned multiple assignment is equivalent to:

$identifier\ 1 = (identifier\ 2 = \ expression)$

That is, right to left nesting of multiple assignments is possible.

An illustration

Let i and j are two integer variables. The multiple assignment expression

$i = j = 3$

will cause the value 3 to be assigned to both i and j. To be more precise, the value 3 is first assigned to the variable j followed by the assignment to i. Similarly, the multiple assignment expression

$i = j = 3.9$

assigns the integer value 3 to both the variables, i and j.

Beside '=' operator, 'C' supports five more assignment operators such as +=, -=, *=, /= and %=. All the five operators are discussed here. Consider the operator +=. The assignment expression

$expression\ 1 \mathrel{+}= expression\ 2$

is equivalent to

$expression\ 1 \ = \ expression\ 1 + \ expression\ 2.$

Similarly,

$expression\ 1 \mathrel{*}= \ expression\ 2$

is equivalent to

$expression\ 1 \ = \ expression\ 1 * \ expression\ 2.$

and so on for all these five assignment operators. Note that expression 1 is usually a variable or an array element.

Example

Consider i and j as integer variables containing values 5 and 7, respectively; and f and g are floating-point variables having values 5.5 and -3.25, respectively. Several assignment expressions making use of these variables are shown below for illustration. Each expression utilises the values of the variables, i, j, f and g.

Expression	Equivalent Expression	Final Value
$i \mathrel{+}= 5$	$i = i + 5$	10
$f \mathrel{-}= g$	$f = f - g$	8.75
$j \mathrel{*}= (i - 3)$	$j = j * (i - 3)$	14
$f \mathrel{/}= 3$	$f = f/3$	1.833333
$i \mathrel{\%} = (j - 2)$	$i = i\%(j - 2)$	0

Each assignment operator has a priority and they are evaluated from right to left based on its priority. The priority of assignment operators is: +=, -=, *=, /= and %=. Assignment operators

have right to left associativity. The unary operations, arithmetic operations, relational and equality operations and logical operations are all carried out before assignment operations.

Example

Consider that x, y and z are integer variables, which have been assigned the value 2, 3 and 4, respectively. The expression

$$x *= -2 * (y + z)/3$$

This expression assigns the value of -8 to the variable, x.

It is interesting to note the order in which the expression is evaluated. You know that the arithmetic operations precede the assignment operation. Therefore, the expression (y+z) will be evaluated first of all thereby producing the value, 7; then this value is multiplied by -2, yielding -14; and this value is further divided by 3 and truncated due to integer division, resulting in -4. Finally, this truncated quotient is multiplied by the original value of x (*i.e.*, 2) to yield the final result of -8.

7.6 Relational Operators

Relational operators are used to compare two values. These operators are used in Boolean conditions or expressions called logical expressions. The resulting expressions will be of the integer type as TRUE is represented by the integer value 1and FLASE is represented by the value 0. The relational operators supported by 'C' are given in Table-7.3.

Table 7.3 'C' Relational operators

Serial Number	Operator	Description
Relational Operators		
1.	<	Less than
2.	<=	Less than or equal to
3.	>	Greater than
4.	>=	Greater than or equal to
Equality Operators		
5.	==	Equal to
6.	!=	Not equal to

The relational operators as shown at serial numbers, 1 to 4 under Table-7.3, fall within the same precedence group and this is lower than the arithmetic as well as unary operators. The associativity of these operators is left to right. There are two equality operators as mentioned at serial numbers 5 and 6 under Table-7.3, which are closely associated with the relational operators. The equality operators fall into a separate precedence group beneath the relational operators. The associativity of these operators is also left to right.

An illustration

Consider that it is required to compare, for equality, the marks obtained by two students in a course on Computer Programming. Suppose, the marks obtained by the first student (denoted by variable, `student_1`) are 50; and that of the second student (denoted by variable, `student_2`) are 47. Now, it is to compare the two for equality. This can be realised by using an appropriate relational operator as follows:

```
student_1==student_2
```

Obviously, the resultant will be FALSE with resultant value as 0.

Example

Suppose i, j and k are integer variables containing values 1, 2 and 3, respectively. Several logical expressions involving these variables are given below:

Expression	Description	Resultant Value
$i < j$	TRUE	1
$(i + j) >= k$	TRUE	1
$(j + k) > (i + 5)$	FALSE	0
$k! = 3$	FALSE	0
$j == 2$	TRUE	1

While carrying out relational and equality operations, operands that differ in type will be converted in accordance with the conversion rules discussed earlier (see Sub-section 7.3.1).

Example

Suppose that i i s an integer variable containing value as 7; f is a floating-point variable having value a 5.5; and c is a character type variable that represents the character 'w'. Several logical expressions showing use of these variables are given below. Each expression comprises of two different type operands. Also, it is assumed that ASCII character set is applicable.

Expression	Description	Resultant Value
$f > 5$	TRUE	1
$(i + f) <= 10$	FALSE	0
$c == 119$	TRUE	1
$c! = 'p'$	TRUE	1
$c >= 10 * (1 + f)$	FALSE	0

7.7 Structure of a Typical 'C' Programme

Every 'C' programme consists of one or more modules called functions. One of the functions must be named as `main`. The programme will always begin with executing the `main` function, which may access other functions. Any other function definitions must be defined separately; either ahead of or after the `main` function (refer to the Module-IV for further details on functions). Each function must contain a function heading, which consists of the function name followed by an

optional list of arguments enclosed in parentheses, a list of argument declarations if arguments are included in the heading and a compound statement, which comprises the remainder of the function.

An illustration

A typical 'C' programme demonstrating various 'C' operators is given below.

```
/* A programme demonstrating 'C' arithmetic operators */
#include < stdio.h>
void main()
{
     int x = 10, y = 20;
     printf("x = %d\n",x);
     printf("y = %d\n",y);
     /* demonstrates = and + operators */
     y = y + x;
     printf("y = y + x; y = %d\n",y);
     /* demonstrates - operator */
     y = y - 2;
     printf("y = y - 2; y = %d\n",y);
     /* demonstrates * operator */
     y = y * 5;
     printf("y = y * 5; y = %d\n",y);
     /* demonstrate / operator */
     y = y / 5;
     printf("y = y / 5; y = %d\n",y);
     /* demonstrates modulus operator % */
     int remainder = 0;
     remainder = y %3;
     printf("remainder = y %% 3; remainder =
     %d\n",remainder);
     /* keeps console screen until a key stroke */
     char key;
     scanf(&key);
}
```

```
Output produced by the above programme
x = 10
y = 20
y = y + x; y = 30
y = y - 2; y = 28
y = y * 5; y = 140
```

```
y = y / 5; y = 28
remainder = y % 3; remainder = 1
```

7.8 Types of Programme Error

There are three basic types of programme error, *viz.*, syntax error; semantic error and logical error. Programme errors are also referred to as programme bugs. The term 'bug' was so coined that one of the earliest programme errors ever detected involved a moth flying into the computer causing short-circuiting of the electronics. Hence, the process of removing programme errors is called debugging. While that goal is laudable, every programmer, no matter how seasoned, writes programmes that contain errors. However, a skilled programmer can detect, isolate and correct programme errors more quickly than a less skilled programmer. Also, experienced programmers do make fewer programming errors simply because of their experience! The lesson here is that you should expect to make a lot of programme errors in the beginning as every beginner does.

View each programme error as a challenge and learn from the experience. Let us take a quick overview of the three types of programme error.

Syntax Errors

The first type of error is a syntax error. You already know that syntax errors are caused when you don't obey the syntax rules. A common syntax rule you might make in the beginning is forgetting to terminate each programme statement with a semicolon.

Logic Errors

Logic errors are those errors that remain after all the semantic and syntax errors have been removed. Usually, logic errors manifest themselves when the result the programme produces doesn't match the result your test data suggest it should produce. Generlly, logic errors are found in the Process. Logic errors occur when you implement the algorithm for solving the problem incorrectly.

The key to fixing logic errors is to be able to reproduce the error consistently. A repeatable logic error is much easier to track down and fix than an error that appears to be occurring randomly.

Semantic Errors

A semantic error occurs when you obey the syntax rules of the language but are using the statement out of context. For example, a sentence in English is expected to have a noun and a verb. Consider the sentence "The dog meowed". This sentence does obey the rules of having a noun and a verb, but the context of the sentence is out of whack. Dogs don't meow; therefore, the context of the statement is incorrect. The error message, I showed you earlier: the name i does not exist in the current context refers to a type of semantic error. There may well be a variable named i defined somewhere in the programme, but it is not currently in scope. That is, you are trying to use i when it is out of scope.

Lesson 8

OPERATORS AND EXPRESSIONS – PART – II
(Logical and Bitwise Operators)

8.1 Logical Operators

Besides the relational and equality operators, 'C' contains two logical operators (also known as logical connectives) as shown in Table-8.1:

Table 8.1 'C' Logical operators

Serial Number	*Operator*	*Description*
1.	&&	AND
2.	||	OR

These operators as mentioned at serial numbers 1 and 2 in Table-8.1 are known as 'logical AND' and 'logical OR', respectively. These operators are employed on logical expressions to combine the individual logical expressions in order to form compound conditions that come out to be either TRUE or FALSE. The outcome of a logical AND operation will be TRUE only if both the operands are TRUE, whereas that of a logical OR operation will be TRUE if either of the two operands is TRUE or if both the operands are TRUE. Thus, the result of the logical OR is FALSE only if both the operands are FALSE. Please mind it carefully that any non-zero value (not restricted to the truth value of 1 only), is considered as TRUE in such cases.

Example 1:

Suppose that i is an integer variable containing value as 7; *f* is a floating-point variable having value as 5.5; and *c* is a character type variable that represents the character '*w*'. Several complex logical expressions based on these variables along with their resultant values are given below.

Expression	Description	Resultant Value
$(i >= 6)\&\&(c == 'w')$	TRUE	1
$(i >= 6)\|\|(c == 119)$	TRUE	1
$(f < 11)\&\&(i > 100)$	FALSE	0
$(c! = 'p')\|\|((i + f) <= 10)$	TRUE	1

The logical operators fall into different precedence groups. Logical AND has a higher precedence than logical OR. However, the logical operators have lower precedence than the equality operators. The associativity is from left to right.

'C' also includes the unary operator '!' that negates the value of a logical expression, *i.e.*, it causes an expression that is originally TRUE to become FALSE and *vice versa*. This operator is referred to as the logical negation or logical NOT operator.

Example 2:

Suppose that *i* is an integer variable containing value as 7; and *f* is a floating-point variable having value as 5.5. Several logical expressions illustrating the use of these variables and the negation operator along with their resultant values are given below.

Expression	Description	Resultant Value
$f > 5$	TRUE	1
$!(f > 5)$	FALSE	0
$i <= 3$	FALSE	0
$!(i <= 3)$	TRUE	1
$i > (f + 1)$	TRUE	1
$!(i > (f + 1))$	FALSE	0

The hierarchy of operators' precedence (highest to lowest) covering arithmetic, assignment, relational and logical operators is summarised in Table-7.2 given in Lesson 7.

Example 3:

Suppose that *i* is an integer variable containing value as 7; *f* is a floating-point variable having value as 5.5; and c is a character type variable that represents the character '*w*'. Several complex logical expressions based on these variables along with their resultant values are given below.

Expression	Description	Resultant Value
$(i + f) <= 10$	FALSE	0
$i >= 6 \&\& c == 'w'$	TRUE	1
$c != 'p' \|\| i + f <= 10$	TRUE	1

Note that these expressions have already been presented earlier within this Section but with pair of parentheses. However, these parentheses are not necessary because of the natural precedence of operators. Thus, the arithmetic operations will automatically be performed prior to the relational or equality operations. Similarly, the relational and equality operations will automatically be accomplished earlier than the logical connectives. For instance, consider the last expression, *i.e.*, c ! = ' p' || i + f <= 10. At first, the addition operation, *i.e.,*i +f is performed followed by the relational comparison, i + f <= 10 then, the equality comparison, *i.e.*, c ! = ′ p′ is carried out; and finally, the logical OR condition is accomplished.

Complex logical expressions comprising of different logical expressions joined together by the logical operators, && and || are evaluated left to right, but only until the overall truth value (TRUE/FALSE) can be established. Hence, a complex logical expression will not be evaluated in its entirety if its value can be established from its constituent operands. For instance, consider a complex logical expression,

error > .0001&& count < 100

If the expression error > .0001 is FALSE, then the second operand count < 100 will not be evaluated, because the entire expression will be considered as FALSE.

On the other hand, suppose the expression is

error > .0001 || count < 100

If the expression error > .0001 is TRUE, then the entire expression will be TRUE. Hence, the second operand will not be evaluated. However, if the expression error > .0001 is FALSE, then the second expression must be evaluated to determine the final truth value of the whole expression.

8.2 Bitwise Operators

'C' includes bitwise operators to manipulate memory at the bit level. These operators are especially useful for writing low level hardware or operating system code where the ordinary abstractions of numbers, characters, pointers, *etc.*, are insufficient. Bit manipulation code tends to be less portable. Recall that a code is said to be portable if it compiles and runs correctly on different types of computers without any modification (*i.e.*, without a programmer's intervention). The bitwise operations are typically used with unsigned types, `int` and `char`. However, 'C' does not specify the difference between a `short int`, an `int` and a `long int`, except to state that:

`sizeof(short int) ≤ sizeof(int) ≤ sizeof(long).`

You will find that these sizes vary from computer to computer, and possibly even compiler to compiler. The sizes do not have to be distinct. That means all the three sizes could be the same, or two of the three could be the same, provided that the above restrictions are held.

Bitwise operators fall into two categories: binary bitwise operators and unary bitwise operators. Binary operators take two operands, while unary operators take only one operand. Bitwise operators, like arithmetic operators, do not change the value of the arguments. Instead, a temporary value is created. This can then be assigned to a variable.

These operators are quite useful for developing embedded software with application to process control and plant automation (since you often need to manipulate individual bits to turn features on or off or to configure your peripherals). As such, it's very important that you understand what these different operators mean and how they function!

8.2.1 Bitwise `AND`, `OR` and `XOR` operators

These operators require two operands and perform bit comparisons. The operator `AND` (`&`) will copy a bit to the result if it exists in both the operands. The following code illustrates this:

```
main()
{
unsigned int a = 60;     /* (60)10 ≡ (0011 1100)2 */
unsigned int b = 13;     /* (13)10 ≡ (0000 1101)2 */
unsigned int c = 0;
c = a & b;                 /* (12)10 ≡ (0000 1100)2 */
}
```

Note that the comments in above code are enclosed as /* . . . */. $(60)_{10}$ denotes that 60 is a decimal number; and $(0011\ 1100)_2$ represents the binary number equivalent to 60. This notion is applicable throughout this section. Also, you should not confuse with the logical AND and bitwise AND operators, *etc*. These are different.

Similarly, bitwise `OR` (`|`) operator will copy a bit if it exists in either operand. It is illustrated with the following code:

```
main()
{
unsigned int a = 60;     /* 60 = 0011 1100 */
unsigned int b = 13;     /* 13 = 0000 1101 */
unsigned int c = 0;
c = a | b;                /* 61 = 0011 1101 */
}
```

Also, bitwise `XOR` (`^`) operator copies the bit if it is set in one operand (but not both). It is illustrated with the following code:

```
main()
{
unsigned int a = 60;     /* 60 = 0011 1100 */
unsigned int b = 13;     /* 13 = 0000 1101 */
unsigned int c = 0;
c = a ^ b;                /* 49 = 0011 0001 */
}
```

8.2.2 Bit shift operators

The bit shift operators, <<, >>, <<=, and >>= can be used for shifting bits left or right. The left operand's value is moved left or right by the number of bits specified by the right operand. For example:

```
main()
{
    unsigned int Value=4;          /*  4 = 0000 0100 */
    unsigned int Shift=2;
    Value = Value << Shift;        /* 16 = 0001 0000 */
```

```
        Value <<= Shift;              /* 64 = 0100 0000 */
        printf("%d\n", Value);        /* Prints 64      */
        return 0;
    }
```

Example 1:

Write a 'C' programme using bitwise operator ^ (XOR) to swap contents of two variables x1 and x2 without the use of a temporary variable.

```
#include <stdio.h>
void main()
{
int x1 = 50;
int x2 = 52;
printf("\nFirst Number = %d\nSecond Number = %d\n", x1, x2);
x1 ^= x2;
x2 ^= x1;
x1 ^= x2;
printf("\nFirst Number = %d\nSecond Number = %d\n", x1, x2);
}
```

Output:

First Number = 50
Second Number = 52

First Number = 52
Second Number = 50

Example 2:

Write a 'C' programme that multiplies any given number by 4 using bitwise operators. [Note: Bitwise operators allow changing the specific bits within an integer. The following programme multiplies any given number by 4 using left shift (<<) bitwise operator.

```
#include <stdio.h>
void main()
{
    long number, tempnum;
    printf("Enter an integer: ");
    scanf("%ld",&number);
    tempnum = number;
    number = number << 2;   /*left shift by two bits*/
    printf("%ld x 4 = %ld\n", tempnum, number);
}
```

Output:

Enter an integer: 2
2 x 4 = 8

Lesson 9

SIMPLE AND COMPOUND STATEMENTS

9.1 Statement

A typical 'C' programme comprises of various statements. A statement is a part of the programme that can be executed by the compiler. Every statement in the programme alone or in combination specifies an action to be performed by the programme. 'C' provides a variety of statements to attain any function with maximum efficiency. Statements fall into three general types:

- An expression statement
- Compound statement
- Control statement

9.2 Expression Statement

An expression statement consists of an expression followed by a semicolon. The execution of an expression statement causes the expression to be evaluated.

Examples:

```
a = 34;
c = a + b;
++i;
```

The first two statements are assignment statements. Each causes the value of expression on the RHS of the 'equal to' sign to be assigned to the variable on the LHS. The third statement is increment statement that increments the value of variable '`i`' by `1`.

9.3 Compound Statement

A compound statement consists of several individual statements enclosed within a pair of braces, `({and})`. Thus, the compound statement provides capability for embedding statements within other statement(s). A compound statement being a group of statements, which are bound together to form a programming logic, are also called a block statement. As you have already seen, a block statement begins with a left curly bracket, `{`, and ends with a right curly bracket, `}`. This is illustrated below with the help of an example code fragment:

```
{
  a = 3; b = 4;
  perimeter  = 2*(a+b);
  area = a*b;
}
```

Example 1: Programme to convert length given in feet to equivalent centimetres. Given that one foot contains 30.48 cm. (See Example-1 of Lesson-2 for pseudo code and flowchart for this problem).

```
#include <stdio.h>
int main()
{
   float len_ft, len_cm;
   printf("Enter length in feet: ");
   scanf("%f", &len_ft);
   len_cm=len_ft*30.48;
   printf("Length in ft and in cm is: %5.2f, %5.2f,
           respectively", len_ft, len_cm);
   return 0;
}
```

Output:

```
Enter length in feet: 2.4
Length in ft and in cm is:  2.40 and 73.15, respectively.
```

Example 2: Programme to calculate the real roots of a quadratic equation. (See Example-2 of Lesson-2 for pseudo code and flowchart for this problem).

```
#include < stdio.h>
#include < math.h>
int main()
{
   float a,b,c,d,x_1, x_2;
   printf("Enter a, b and c: ");
   scanf("%f %f %f", &a, &b, &c);
   d=sqrt(b*b-4×a×c);
   x_1=(-b+d)/(2*a);
   x_2=(-b-d)/(2*a);
   printf("a, b, c, x_1, x_2: %5.2f, %5.2f, %5.2f,
          %5.2f,%5.2f",a, b, c, x_1, x_2);
   return 0;
}
```

Output:

```
Enter a, b and c: 2 3 1
```

```
a, b, c, x_1, x_2:  2.00,  3.00,  1.00,-0.50, -1.00
```

Example 3: Programme to calculate area and perimeter of a circle when its radius is known.

```
#include < stdio.h>
int main()
{
    float area, pi, perimeter, radius;
    printf("Enter radius of circle: ");
    scanf("%f", &radius);
    {
       pi=3.14;
       area = pi*radius*radius;
       perimeter = 2.0*pi*radius;
    }
    printf("\nArea of circle is: %f", area);
    printf("\nPerimeter of circle is: %f",
           perimeter);
    getch();
    return 0;
}
```

Output:

```
Enter radius of circle: 3
Area of circle is: 28.260000
Perimeter of circle is: 18.840000
```

9.4 Control Statements

Control statements are used to control the flow of execution of the instructions written in a programme. Normally, the instructions are executed sequentially, *i.e.*, one after the other or in the same order as they appear in the programme. This type of control structure is known as sequence control structure and there is neither any decision making process involved nor do they require repeated execution of group of statements. However in normal routine, it is often required to change the order of execution of statements. Hence, besides sequence control statements, several other control structures are required to accomplish a real-life problem-solving task. These structures are:

- Selection control structure (decision or branching or conditional statements) – These structures decide the flow of statements based upon the results of evaluation of pre-define conditions, *e.g.*, `if…then…else` and `switch…case` statements (discussed in Lesson 10) come under this category.
- Repetition control structure (looping or iteration statements) – These structures are used to run a particular block of statements repeatedly (*i.e.*, in a loop), *e.g.*, `for`, `while…do` and `repeat…until` statements (discussed in Lesson 11) come under this category.
- Jumping statements (`goto` statement)
    - Jump Statements – These are used to make the flow of your statements from one point to another, *e.g.*, `break`, `continue`, `goto` and `return` statements (discussed in Lesson 11) belong to this category.
    - Label Statements – These are used as targets/way-points for jump and selection statements. `case` (discussed with switch in Lesson 11) and label (discussed with goto) come under this category

All types of control statement are discussed in detail in the following lessons 10 and 11 of this Module III.

9.5 Comments Statement

Besides various executable (that are compiled by compiler) as well as non-executable statements (*i.e.*, processor directive statements such as the `define` statement) as discussed earlier, there is another non-executable statement supported by 'C', *i.e.*, comments statement that is ignored by the compiler while compiling the source code to generate object code leading to the executable code. Comments provide the programmers with information about the code. Good commenting generally increases the productivity of your team as you do not have to go back every time to figure out what a particular function/segment is doing; you can read the comments instead. Remember that while your source code will be compiled and optimised by a machine; it would always be read and modified by a human being! Hence, comments statement is important and is discussed below.

Any text or statement in a 'C' source code, which is embedded within the symbols, '`/*`' and '`*/`' is treated as a comment statement, which is non-executable and just ignored by the compiler while compiling the code. The comment statement can occur within expressions, and can span multiple lines. Proper care must be taken by the programmer to mark the beginning and end of the comment using appropriate terminators meant for the purpose. Consider the following code segment and note the associated side effects arising due to improper use of comment statement terminators:

```
1. /* This line will be ignored.
2. /*
3. These lines will also be ignored. Note that the comment opening token above did not start a new comment, and
4. */
5. Hence, this line and the line below it will not be ignored. Both may produce compile-time errors!
6. */
```

C++ style line-comments start with `//` and extend to the end of the line. This style of comment originated in Basic Combined Programming Language (BCPL) and became valid 'C' syntax in C99; however, it is neither available in the original 'C' nor in the ANSI 'C'. This is illustrated in following code snippet:

```
// this line will be ignored by the compiler
/* these lines
```

```
will be ignored
by the compiler */
x = *p/*q;  /* note: this comment starts after the 'p' */
```

Lesson 10

DECISION CONTROL STATEMENTS

10.1 Decision Control Statements

A programme consists of a number of statements, which are usually executed in a particular sequence. Programming can be made powerful, if one can control the order in which statements are performed. Generally, this is achieved with decision control statement (also known as selection statements). These statements are used to create special programme features such as decision making or branching structures (*i.e.*, logical tests). In a nutshell, a decision control statement makes decision about the next course of action depending upon different options and or conditions. 'C' supports a variety of decision control statements as described below.

10.2 `if`…`else` Statement

The `if…else` statement is used to decide whether to perform an action at a special point or to decide between two courses of action. The general syntax is as follows:

```
if (conditional expression)statement;
```

This can be extended with `else` option to choose a course of action out of two available options as follows:

```
if (conditional expression)statement_1;
else statement_2;
```

Thus, if the underlying expression comes out to be TRUE, the `statement_1` is executed, otherwise `statement_2` is performed.

Illustration

The following code snippet tests whether a student passed an examination with the passing marks as 60.

```
if (marks >= 60)
        printf("Pass\n");
else
        printf("Fail\n");
```

In this example, within `if` statement, the underlying conditional expression is evaluated and for its truth value, *i.e.*, TRUE or FALSE. If it is TRUE, the message '`Pass`' is displayed on output device, otherwise `else` statement is executed and it displays '`Fail`' as the output.

In case there are multiple statements to be performed depending on the truth value of the conditional expression, they are embedded in curly brackets, {}.

Illustration

```
if (marks >= 60)
{       printf("Congratulations\n");
        printf("You have passed the
examination\n");
}
else
{       printf("Work hard\n");
        printf("You failed the examination\n");
  }
```

10.3 Nested `if`…`else` Statement

Sometimes, you have to make a multi-way decision based on several conditions. This is achieved by cascading several comparisons; and depending upon the result of these multiple comparisons, the following statement or block of statements is executed. The moment, one of these comparisons gives a TRUE value, no further comparisons are performed.

Illustration

Assume that grades are awarded to the students depending on the examination results. Grade 'A' is given to the students having marks greater than or equal to 80; Grade 'B' to the students securing marks (equal to or more than 60 but less than 80); Grade 'C' for the marks (equal to or greater than 45 but below 60) and students attaining less than 45 marks are considered to be 'failed'.

```
if (marks >= 80)
        printf("Passed: Grade A\n");
else if (marks >= 60 && marks < 80)
        printf("Passed: Grade B\n");
else if (marks >= 45 && marks<60)
        printf("Passed: Grade C\n");
else
        printf("Failed\n");
```

In this example, all comparisons test a single variable, *i.e.*, `marks`. First step is to check whether 'marks are greater than or equal to 80', if this condition comes out to be TRUE, it prints the message 'Passed: Grade A' and exits the `if` statement (to take up next statement in programme); otherwise it moves onto the next `else if` statement, *i.e.*, the condition 'the marks lie between 60 and 80', if it is found TRUE, the output will be 'Passed: Grade B' and exits the `if` statement; otherwise, it further goes to the next `else if` statement to check whether 'marks lie between the range, 45 to 60', if this condition is found TRUE, the message 'Passed: Grade C' is printed and exits the `if` statement; otherwise the message 'Failed' is printed finally and exits the `if` statement.

10.4 `switch` and `break` Statements

10.4.1 `switch` statement

The `switch` statement is another form of the multi-way decision-making statement. It is well structured, but can only be used in certain cases where only one variable is to be tested and all branches must depend on the value of that variable. The variable must be an integer type or character type. In `switch` statement, different `cases` are presented and are checked one by one to see if one `case` matches the value of the underlying `constant expression`. If a `case` matches, its block of statements is executed. If none of the cases match, the `default` code is executed. The general syntax of `switch` statement is:

```
switch (expression)
  {
    case constant_1:
       statement;
       break;
    case constant_2:
       statement;
       break;
       .
       .
       .
   default:
       statement;
  }
```

10.4.2 `break` statement

`break` statement is used to force an early exit from a loop (discussed in next Lesson-11) or a `switch`; hence, control passes to the first statement beyond the loop or a `switch`. With loops, `break` can be used to exit from the loop, or to implement a loop with a test to exit in the middle of the loop body. A `break` within a loop should always be protected within an '`if`' statement, which provides the test to control the exit condition.

Illustration

Consider addition and multiplication of two numbers using `switch` statement.

```
int a, b;
int ch;
scanf("%d %d %d", &a, &b, &ch);
switch(ch)
 {
  case 1:
     result = a+b;
     break;
```

```
        case 2:
            result =a*b;
            break;
        default:
            printf("Wrong Choice!");
    };
```

Here, ch is taken as int type constant expression. The value of this expression decides as to which case is to be executed, *i.e.*, if the value of the expression, ch comes out to be 1, the case 1 will be executed leading to addition of the two numbers and break statement prevents any further statements contained within the switch statement, from being executed by leaving the switch; and if ch comes out to be 2, the case 2 statements are executed thereby computing multiplication of the two numbers, and break statement will leave switch statement. If the expression, ch assumes any value other than 1 and 2, the default code is executed, displaying the message "Wrong Choice!" on the output device.

10.5 goto Statement

'C' has a goto statement, which permits unstructured jumps to be made. The goto statement is used to alter the programme execution sequence by transferring the control to some part of the programme. The general syntax is:

```
goto statement lable;
```

where lable is valid 'C' identifier used to label the destination. There are two ways of using the goto statement:

- Unconditional goto
- Conditional goto

10.5.1 Unconditional goto statement

The unconditional goto statement is used to transfer the control from one programme part to other part without checking any condition.

Illustration 1

```
{
    Begin : printf("Welcome!");
    goto Begin;
}
```

Here, Begin is a label. First of all, printf function is executed and then the control moves onto the goto statement, which further passes the control to the label, Begin without any condition and this process repeats. Hence, an infinite loop is set up, which keeps on printing the message 'Welcome!' repeatedly! It can be stopped externally.

Illustration 2

In a difficult programming situation, it seems so easy to use a goto statement to take the control where you want. You can understand the use of goto keyword with the help of the following example:

```
main( )
{
  int goals ;
  printf("Enter the number of goals scored against India ")
;
  scanf("%d", &goals) ;
  if (goals<=5)
    goto sos;
  else
  {
    printf("About time soccer players learnt C\n");
    printf("and said goodbye! Goodbye! to soccer");
    exit(); // stops programme execution
  }
  sos:
  printf("To err is human!");
}
```

Test Output

```
Enter the number of goals scored against India 10
About time soccer players learnt C
and said goodbye! Goodbye! to soccer
```

Explanation

If the underlying condition is fulfilled, the goto statement transfers control to the label 'sos', causing printf function statement labelled 'sos' to be executed. Note the following points:

- The label can be on a separate line or on the same line as the statement following it, as in, `sos: printf ("To err is human!");`
- Any number of goto statements can take the control to the same label.
- The exit() function is a standard library function, which terminates the execution of the programme. It is necessary to use this function since we don't want the statement: `printf ("To err is human!");` to get executed after execution of the else block.
- The only programming situation in favour of using goto is when we want to take the control out of the loop that is contained in several other loops.

10.5.2 Conditional goto statement

The conditional goto statement passes the control of execution from one part of the programme to other, depending upon some condition case.

Illustration: Consider the following code segment.

```
int a = 9, b = 3;
if(a > b)
     goto out_1;
else
     goto out_2;
out_1: printf(" a is largest");
return;
out_2: printf(" b is largest");
```

Here, a and b are two variables of integer type. The 'if' statement compares a and b; obviously, in this case, the resultant value comes out to be TRUE; accordingly, the goto statement passes the control to the label, out_1; otherwise, the control will be transferred to the label, out_2. The students may like to interchange the values of a and b in the above code segment (*i.e.*, a = 3, b = 9) and try to re-compile and run the same to see the effect of 'else' part of the aforementioned 'if' statement; in this case, the control will be sent to the out_2 label.

Example 1: Programme to calculate tax for a given amount of salary according to pre-defined criteria as specified in the algorithm given under Exercise-1 of Lesson-1.

```
/*  A 'C' program for the algorithm given in Exercise 1,
Lesson 1  */
#include <stdio.h>
void main()
{
 long int salary;
 long float tax;
 printf("\nEnter salary : ");
 scanf("%ld", &salary);
 if (salary < 50000)
    tax = 0;
 else if (salary > 50000 && salary< 100000)
          tax=50000*0.05;
 else
          tax=100000*0.30;
 printf("\nTax : %lf", tax);
}
```

Output:

```
Enter salary : 100267
```

```
Tax : 30000.000000
```

Example 2: Let us see what output the following typical code produces upon its execution?

```
#include<stdio.h>
void main()
{
    int m=5,n=10,q=20;
    if(q/n*m)
        printf(" Sree Rama");
    else
      printf(" Sree Krishna");
    printf(" Mahadev Shiv");
}
```

Output:

```
Sree Rama Mahadev Shiv
```

Explanation

Consider the following expression:

```
q / n * m
```

In this expression there are two operators, *viz.*, / (division operator) and * (multiplication operator). You know both the operators have same precedence and associativity from left to right. Thus, in this case, associativity decides, which operator executes first. Accordingly, the division operator (/) is executed first followed by multiplication (*) operator, as follows:

```
  q / n * m
= 20 / 10 * 5
  = 2 * 5
    =10
```

Recall that in 'C', zero represents `FLASE` and any non-zero number represents `TRUE`. Since the resultant value in above case comes out to be 10, which is a non-zero number; therefore, '`if`' clause will execute and print: `Sree Rama`.

Now, note that in the '`else`' clause above, there are no opening and closing curly brackets. Therefore, compiler considers only one statement as a part of the '`else`' clause. Thus, last statement, *i.e.*,

```
printf(" Mahadev Shiv");
```

is not part of `if…else` statement. Hence, at the end compiler also prints: `Mahadev Shiv`. Therefore, the output of above code is:

```
Sree Rama Mahadev Shiv
```

Example 3: What output the following code would produce when executed?

```
#include<stdio.h>
void main()
{
    if(!printf("Sree Rama"))
        if(printf(" Sree Krishna"));
}
```

Output:

```
Sree Rama
```

Explanation:

Note that the `printf` statement is a kind of function (discussed later) and every function generally returns some value as an outcome. The return type of `printf` function is `int`. That is., this function returns an integral value, which is equal to the number of characters a `printf` function will print on the screen. Thus, in the present example, first of all, `printf` function will print: `Sree Rama`. Since, it is printing 9 characters; therefore, it will return value as 9. Hence, the '`if`' condition, `!printf("Sree Rama")` comes out to be equivalent to `!(9)`, which is equal to `0`. As stated earlier, in 'C', zero represents FALSE. Thus, `if(0)` is FALSE in present case; accordingly, the next statement embedded inside the body of the first '`if`' statement will not execute.

Lesson 11

LOOP CONTROL STATEMENTS – PART I

(`while`, `do...while` and `for` Loops)

11.1 Loops

Besides decision control statements as described in previous Lesson-10, the other main types of control statement are the iterative statements that are used to create looping mechanism. Thus, loops allow a statement (or block of statements) to be repeated until a given criterion is fulfilled. 'C' facilitates three types of loop:

a) `while`
b) `do…while`
c) `for`

11.2 `while` Loop

The `while` loop keeps repeating an action until a condition gets `FALSE`. This looping statement is useful where the programmer does not know beforehand as to how many times the loop will be executed. The general syntax is:

```
while (expression)
{
    Statements;
}
```

Example 1: Programme to generate even series from `1` to `50` using `while` loop.

```
int i = 2;
while (i <= 50)
{
     printf("%d\t", i);
     i = i + 2;
}
```

In this example, '`i`' considered as an integer type loop-variable, which is initialised with value, '`0`'. In the `while` loop, first of all the underlying criterion is checked, *i.e.,* whether the value of '`i`' is less than or equal to 50; if TRUE, the control will enter in the loop and keeps on updating the loop-variable '`i`' and prints the output. Once the condition gets FALSE, the control will pass to the next statement out of the loop. The above code segment will print the following result:

```
2     4     6     8     10    12    14    16    18    20
22    24    26    28    30    32    34    36    38    40
42    44    46    48    50
```

Example 2: Programme to evaluate the given series, $cos(x) = 1 - \frac{x^2}{2} + \frac{x^4}{4!} - \frac{x^6}{6!}$...; to calculate $\cos(x)$.

Also, it compares the calculated value with the one by using inbuilt library function.

```
//C programme to find sum of cosine series?
#include <stdio.h>
#include <conio.h>
#include <math.h>
void main()
{
  clrscr();
  float power = 2.0, numr, denr = 1.0, x1, sum;
  int i = 1, n, s = -1, x;
  printf("\n\n\t ENTER THE ANGLE...: ");
  scanf("%d", &x);
  x1 = 3.142 * (x / 180.0);
  sum = 1.0;
  numr = x1*x1;
  printf("\n\t ENTER THE NUMBER OF TERMS...: ");
  scanf("%d",&n);
  while(i<=n)
  {
    denr = denr * power * (power - 1.0);
    sum = sum + (numr / (denr * s));
    s = s * (-1);
    power = power + 2.0;
    numr = numr * x1 * x1;
    i++;
  }
  printf("\n\t THE SUM OF THE COSINE SERIES IS : %0.5f", sum);
  printf("\n\n\t THE VALUE OF COSINE FUNCTION FOR ANGLE = %d IS :
  %0.5f",x,cos(x1));
  getch();
}
```

Let us input the angle as `5` radians and number of terms for iteration as `10`. Then, the above programme produces the following output:

```
    ENTER THE ANGLE...: 5

    ENTER THE NUMBER OF TERMS...: 10
```

```
THE SUM OF THE COSINE SERIES IS : 0.99619

THE VALUE OF COSINE FUNCTION FOR ANGLE = 5 IS : 0.99619
```

Example 3: Programme corresponding to the algorithm given in Example-1(b) under Lesson-1 to compute average of n numbers using `while` loop.

```
#include <stdio.h>
void main()
{
 float x, sum=0.0;
 int n, i=1;
 printf("Enter value for n: ");
 scanf("%d",& n);
 while (i<=n)
 {
   printf("Enter value for x: ");
   scanf("%f",& x);
   sum=sum+x;
   i=i+1;
 };
 mean = sum/ n;
 printf("\nMean =%5.2f", mean);
}
```

Test Output:

```
Enter value for n: 5
Enter value for x: 10
Enter value for x: 10.5
Enter value for x: 12.12
Enter value for x: 13.45
Enter value for x: 23.90
Mean=13.99
```

Example 4: Programme corresponding to the algorithm given in Example-2, Lesson-1 to input an examination's marks of *n* students as well as to test each student's marks for the award of a grade.

```
#include <stdio.h>
void main()
{
 int n, marks, i=1;
 printf("\nEnter value for n : ");
 scanf("%d",& n);
```

```
 while (i<=n)
 {
    printf("\nEnter eaxm's marks : ");
    scanf("%d",& marks);
    if (marks >= 80)
       printf("\nDistinction");
    else if (marks >= 60 && marks < 80)
       printf("\nMerit");
    else if (marks >= 40 && marks < 60)
       printf("\nPass");
    else if (marks < 40)
       printf("\nFail");
    i=i+1;
 }
}
```

Test Output:

```
Enter value for n : 5
Enter eaxm's marks : 92
Distinction
Enter eaxm's marks : 67
Merit
Enter eaxm's marks : 56
Pass
Enter eaxm's marks : 32
Fail
Enter eaxm's marks : 71
Merit
```

11. 3 `do…while` Loop

The `do…while` loop is very similar to the `while` loop except that the condition is tested at the end of the loop body. This ensures that the loop is executed at least once. Provided that the underlying criterion is TRUE, the loop statement(s) are repeated. The general syntax is:

```
do
{
   statements;
}
while (expression);
```

Note that the statements underlying a `while` loop might never be executed if the expression is FALSE; however, a `do…while` statement will always execute the statements contained within the body of the loop at least once.

Illustration – Consider the following code segment:

```
int  i = 1;
do
{
    printf("I like computer programming!");
    ++i;
}
while (i <= 7);
```

In this code segment, the statements inside the loop are executed at least once without any consideration for the underlying criterion; as a result the slogan "`I like computer programming`" is printed during the first iteration. Thereafter, the conditional expression is evaluated; and if it is found to be FALSE, it will not execute the block of statements any more and exits the loop thereby transferring the control to the next statement.

Example 5: Programme to calculate the resultant focal length : f , when f_1 and f_2 are placed in contact; use formula;

$$f = \frac{(f_1+f_2)}{(f_1 f_2)}$$

compute for following pairs of focal lengths:

$f_1 = -10, -8, -6, 0, \ldots, +8, +10$;

$f_2 = -0.5, -0.4, \ldots, +0.4, +0.5$

```
#include<stdio.h>
#include<conio.h>
void main()
{
float f=0.0, f1=-10.0, f2=-0.5;
clrscr();
do
{       if(f1!=0.0&&f2!=0.0)
        {
          f=(f1+f2)/(f1*f2);
          printf("\nf1, f2 and f values are %5.2f, %5.2f
                  and %7.3f",f1, f2, f);
          f1=f1+2.0;
          f2=f2+0.1;
        }
```

```
            else
            {
              f1=f1+2.0;
              f2=f2+0.1;
              f=(f1+f2)/(f1*f2);
    printf("\nf1, f2 and f values are %5.2f, %5.2f
                      and %7.3f",f1, f2, f);
              f1=f1+2.0;
              f2=f2+0.1;
            }
    }while(f1!=12.0&&f2!=0.6);
    getch();
    }
```

Example 6: Programme equivalent to algorithm described in Example-1(c) under Lesson-1 to compute average of n numbers using `do…while` (`repeat…until`) loop.

```
#include <stdio.h>
void main()
{
 float x, sum=0.0;
 int n, i=1;
 printf("Enter value for n: ");
 scanf("%d",& n);
 do
 {
   printf("Enter value for x: ");
   scanf("%f",& x);
   sum=sum+x;
   i=i+1;
 } while (i<=n);
 mean = sum/ n;
 printf("\nMean =%5.2f", mean);
}
```

Test Output:

```
Enter value for n: 5
Enter value for x: 10
Enter value for x: 10.5
Enter value for x: 12.12
Enter value for x: 13.45
Enter value for x: 23.90
Mean=13.99
```

11.4 `for` Loop

The `for` loop is useful when the number of iterations of the loop is known before the loop is executed. The head of the '`for`' loop comprises of three constituents separated by semicolons. The general syntax is:

```
for (initial statement; conditional expression; loop
statement)
{
     body of the loop ;
}
```

- The `initial statement` is used to initialise the loop variable
- The `conditional expression` is used to check whether or not the loop is to be continued again
- The `loop statement` is used to change the loop-variable value for further iteration.

Note that all the three constituents are optional; hence, the following typical `for` statement is valid that forms an infinite loop to display the message "`Hello, this is an infinite loop!`" repeatedly, as the conditional expression is always considered as TRUE, if the same is absent).

```
for (;;)
{
  printf("Hello, this is an infinite loop!\n");
}
```

This will keep on executing the underlying `printf` statement forever until interrupted by some other means. This might be done so by incorporating either a `return` or a `break` statement inside the loop after `printf` statement.

Example 7: Programme corresponding to the algorithm given in Exercise-7 under Lesson-1?

```
/*    c programme for Exercise-7, Lesson-1      */
#include <stdio.h>
void main()
{
   int n;
   for (n = 1; n <= 4; n++)
   printf("\n\nLoop%d", n);
}
```

Output:

```
Loop1
Loop2
```

```
Loop3
Loop4
```

Example 8: Programme to write a table of 2 using for loop.

```
#include <stdio.h>
void main()
{
                int a=0;
                int i;
                for ( i=1; i<=10; i++)
                {
                   a = 2 * i;
                   printf(" %d",a);
                }
}
```

Here, 'a' and 'i' are int type variables. Within for loop first step is to initialise 'i' with value '1'and then condition is checked that its value is less than or equal to '10', if TRUE, statements within the body of the loop are executed and the value of 'i' is incremented by '1'. Once again, the condition is checked; if it is found valid, the statements within the body of the loop are executed and so on until the condition becomes FALSE. As a result, the following table is displayed on the output device:

```
2      4      6      8      10      12      14      16      18      20
```

Example 9: Programme equivalent to the algorithm given in Example-1(d) under Lesson-1 to compute average of any ten numbers using for loop.

```
/*    c programme to sum 10 numbers using for loop*/
#include <stdio.h>
void main()
{
int i;
float x, mean, sum = 0.0;
for  (i=1; i<=10; i++)
{
   printf("\nEnter value for x: ");
   scanf("%f", &x);
   sum = sum + x;
}
mean = sum/10.0;
printf("\nMean =%5.2f", mean);
}
```

Test Output:

```
Enter value for n: 5
Enter value for x: 10
Enter value for x: 10.5
Enter value for x: 12.12
Enter value for x: 13.45
Enter value for x: 23.90
Mean=13.99
```

LESSON 12

LOOP CONTROL STATEMENTS – Part II

(Nesting of Control Statements)

12.1 Nesting Decision and Loop Control Statements

Nesting means embedding one object in another object. Nesting is quite common in 'C' structured programming, where different logic structures, *viz.*, sequence, selection and repetition are combined (*i.e.*, nested in one another), usually indicated through different indentation levels within the source code as delineated through the following code segments:

Illustration 1:

```
main()
{
  int row, col, sum ;
  for (row = 1; row <= 3; row++) /* outer loop */
  {
    for (col = 1; col <= 2; col++) /* inner loop */
    {
       sum = row + col ;
       printf ("row = %d column = %d sum = %d\n", row, col,

                sum);
    }
  }
  return 0;
}
```

Output:

```
row = 1 column = 1 sum = 2
row = 1 column = 2 sum = 3
row = 2 column = 1 sum = 3
row = 2 column = 2 sum = 4
row = 3 column = 1 sum = 4
row = 3 column = 2 sum = 5
```

Explanation

In this illustration, for each value of the variable `row` in the inner loop is iterated twice, with the variable `col` taking on values from `1` to `2`. The inner loop terminates when the value of `col` exceeds `2`, and the outer loop terminates when the value of `row` exceeds `3`. Evidently, the body of the outer for loop is indented, and the body of the inner `for` loop is further indented. These multiple indentations make the programme reading easy for better understand the logic. Instead of using two statements, one to calculate `sum` and another to print it out, we can integrate this into a single statement as follows:

```
printf ("row = %d column = %d sum = %d\n", row, col, row+col);
```

The way `for` loops have been nested here, similarly, two `while` loops can also be nested. Also, `for` loop is possible to be nested within a `while` loop and *vice-versa*

Illustration 2: The following code segment demonstrates nesting of decision control structure within loop control structure.

```
int i;
for(i=1;i<=10;i++)
{
if(i==5)
   break;
else
   printf("%d", i);
}
printf("Hello");
```

Output:

```
1234Hello
```

In this illustration, within the '`for`' loop, `int` type variable '`i`' is initialised with value as '`1`' having upper limit '`10`'; and is incremented by `1` when the underlying condition is satisfied. In the '`if`' statement, the condition is checked that `i==5` or not, if `TRUE`, the '`break`' statement is executed and control passes to the next statement out of the loop, showing the output: `Hello`. Otherwise, the '`else`' statement will be executed thereby printing the value of '`i`'; followed by incrementing value of '`i`' by `1` and so on until value equals to `5` or condition gets `FALSE`.

Example 1: Ohm's law is $I = V/R$; write a programme to calculate *I* from given *n* sets of *V* and *R*.

```
#include <stdio.h>
#include <conio.h>
void main()
{
   clrscr();
   int n, k;
```

```
   float i, v, r;
   printf("\nEnter number of sets ");
   scanf("%d",& n);
   for (k=1; k<=n; k++)
   {
     printf("\nEnter values for potential differenc(V) and
            resistance (R) ");
     scanf("%f %f", &v, &r);
     i=v/r;
     printf("\n V = %5.2f;  R = %5.2f; and I =
             %0.2f.", v, r, i);
     printf("\n");
   }
   getch();
}
```

Example 2: Write a programme, which can input a positive integer (<=10000000) and prints it in reverse order, for example, 9875674 to 4765789.

```
#include <stdio.h>
#include <conio.h>
void main()
{
  clrscr();
  long int num, mod, rev=0;
  printf("Enter the number");
  scanf("%ld", & num);
  while(num>0)
  {
    if(num<=10000000)
    {
       mod=num%10;
       rev=(rev*10)+mod;
       num=num/10;
    }
    else
    break;
  }
  printf("Reverse Number is : %ld", rev);
  getch();
}
```

Output:

```
Enter the number 9875674
Reverse Number is : 4765789
```

An exercise for students

Write a programme to calculate sum of squares of all odd integers between *17* and *335*; exclude integers divisible by *7*.

Example 3: Programme corresponding to the algorithm described in Example-3 under Lesson-1 to find and print the largest number among any n given numbers.

```
/*    c program for Example 3, Lesson 1      */
#include <stdio.h>
void main()
{
 int n, current_number, next_number, max, counter=1;
 printf("\nEnter values for n and current number: ");
 scanf("%d %d", &n, &current_number);
 max = current_number;
 while (counter < n)
 {
     counter = counter + 1;
     printf("\nEnter next number: ");
     scanf("%d", &next_number);
     if(next_number > max)
        max = next_number;
 }
   printf("\nThe largest number is : %d", max);
}
```

Test Output:

```
Enter values for n and current number: 5 30
```

```
Enter next number: 90
Enter next number: -1
Enter next number: 99
Enter next number: 87
The largest number is : 99
```

Example 4: Write a programme equivalent to the algorithm given in Exercise-8 under Lesson-1 to calculate sum of squares of a given set of numbers; and test the programme with input values for the variables `n` and `val` as `5`; and `6, 12, 24, 48` and `96`, respectively.

```
/*  'C' programme for Exercise-8 of Lesson-1  */
#include <stdio.h>
void main()
{
 int n, i=1;
 long float val, ssq=0;
 printf("\nEnter n : ");
 scanf("%d", &n);
 while (i<=n)
 {
     printf("\nEnter val : ");
     scanf("%lf",& val);
     ssq = ssq + val*val;
     i=i+1;
 }
 printf("\nSum of squares :%lf", ssq);
}
```

Output:

```
Enter n : 5
Enter val : 6
Enter val : 12
Enter val : 24
Enter val : 48
Enter val : 96
Sum of squares :12276.000000
```

Example 5: Programme to check whether a given number is prime number?

You know that a prime number is a natural number, which is divisible by `1` and itself.

```
#include <stdio.h>
#include <conio.h>
void main()
{
  int num,i,z;
  printf("Enter the value of num");
  scanf("%d",&num);
  i= 2;
  while (i < num)
  {
    if(num%i==0)
    {
      z=1;
  }
  i=i+1;
}
printf("After checking the result is\n");
if(z== 1)
{
printf("Given number is not prime number");
}
else
{
printf("Given number is prime number");
}
getch();
}
```

Test Output

```
Enter the value of num 7
After checking the result is
Given number is prime number
```

12.2 `continue` Statement

The `continue` statement is just opposite to the `break` statement, *i.e.*, unlike the `break` statement that terminates the loop, the `continue` statement immediately loops again thereby skipping the rest of the code. It works only within loops where its effect is to force an immediate jump to the loop control statement. Like a `break`, `continue` should be protected by an '`if`' statement.

Illustration

Consider the following code segment to comprehend the `continue` statement.

```
int i;
for (i = 1;i <= 10; i++)
{
     if(i==5)
          continue;
     else
          printf("%d ", i);
}
printf("\nNote that fifth value of i is not printed!");
```

The above code displays the following information (which is self explanatory) on the output device:

```
                1   2   3   4   6   7   8   9   10
Note that fifth value of i is not printed!
```

Lesson 13

FUNCTIONS IN 'C' – PART I

(Declaration, Function Prototypes, Parameter Passing and Function Access)

13.1 Functions

A function is a self-contained block of statements, which performs some specific, well-defined task and always returns a single value to the calling function. Every 'C' function comprises of one or more functions; one of which must be defined as `main`. The programme execution begins with carrying out the instructions contained in the `main` function. All other functions, if any, being a part of the programme will be subordinate to the `main` function. If the programme contains several functions, their definitions may appear in any order, though they must be independent of each other. A function can be accessed or called from different points in a programme. Generally, a function will process the information passed through the calling point of the programme and returns a single value as a result. This information is passed to the function through special identifiers called arguments or parameters and the function output (a single value) is returned by means of the `return` statement. Thus, once the function has completed its intended task, the control automatically reaches back to the calling point. On the other hand, some functions accept information but do not return anything, *e.g.*, library function `printf`, while other functions such as `scanf` return multiple values. Evidently, functions are used to minimise the repetition of code. There are two types of function in 'C' as described below.

13.2 Programmer-defined Function Declaration, Parameter Passing and Function Access

A function written by a programmer is called programmer-defined function. The function definition has two principal components; the first line that includes the argument declaration and the body of the function. The general syntax of the function statement is:

datatype name(*type 1 arg_1, type 2 arg_2, …, type n arg_n*)

where *datatype* represents the data type of the resultant value returned by the function after execution of the instructions contained in the body of the function; *name* represents the function name for reference by other functions; and *type 1*, *type 2*, …, *type n* are the data types of the arguments *arg_1*, *arg_2*, …, and *arg_n*. Note that the data types are assumed to be of the type `int` if they are not declared explicitly.

An illustration

The following code segment demonstrates as to how to define and access functions in a programme. Note that `add` is the function that computes sum of two integer numbers supplied through the `main` function while calling the same. Always, remember that the arguments and the corresponding actual variables should be same in type, order and number.

```
int add (int x, int y)
```

```
{
        int z;
        z = x + y;
        return (z);
}
void main()
{
        int i, j, k;
        i = 15;
        j = 5;
        k = add(i, j);
        printf ("The value of k is %d\n", k);
}
```

Output

```
The value of k is 20.
```

13.3 Function Prototypes

In the above programme, you have seen that the programmer-defined function preceded the `main` function. Thus, when this programme is compiled the programmer-defined function will have been defined before the first access. However, many programmers prefer a top-down approach, in which the `main` function appears ahead of the programmer-defined function. In such situations, the function access (from within the main) will precede the function definition. This sounds confusing to the compiler, unless the compiler is first instructed that the function being accessed will be defined later in the programme. A function prototype is used for this purpose. The function prototype is usually written at the beginning of the programme, *i.e.*, ahead of any programmer-defined function including the `main` function. The general syntax of the function prototype is:

datatype name(*type 1 arg_1, type 2 arg_2, …, type n arg_n*);

where *datatype* represents the data type of the resultant value returned by the function after execution of the instructions contained in the body of the function; *name* represents the function name for reference by other functions; and *type 1, type 2, …, type n* are the data types of the arguments *arg_1*, *arg_2*, …, and *arg_n*. Note that the data types are assumed to be of the type `int` if they are not declared explicitly. You might have noticed that the function prototype resembles the first line of a function definition statement (though, a function prototype ends with a semicolon!). Also, note that the names of the arguments within the function prototype need not be declared elsewhere in the programme, since these are dummy arguments that are recognised only within the prototype. In fact, the argument names can be omitted; however, the argument data types are essential. Function prototypes are not essential but desirable as they facilitate error checking between the calls to a function and the corresponding function definition.

An illustration

The programme discussed in the previous section is revised to demonstrate the effect of the function prototype as follows:

```
int add (int x, int y);
void main()
{
      int i, j, k;
      i = 15;
      j = 5;
      k = add(i, j);
      printf ("The value of k is %d\n", k);
}
int add (int x, int y)
{
      int z;
      z = x + y;
      return (z);
}
```

Output

```
The value of k is 20.
```

13.4 Scope of Function

Only a limited amount of information is available within the body of each function. Variables declared within the calling function cannot be accessed from the outside functions unless they are passed to the called function as arguments.

13.5 Global Variables

A variable that is declared outside all functions is called global variable. Global variables do not die on return from a function. Their value is retained and is available to any other function within the whole programme.

13.6 Local Variables

A variable that is declared within a function is called local variable. They are created each time the function is called and destroyed on return from the function. The values passed to the functions (arguments) are also treated like local variables.

13.7 Static Variables

Static variables are like local variables but they do not die on return from the function. Instead, their last value is retained and it becomes available when the function is called again.

13.8 Parameter Passing in 'C'

The function is a self contained block of statements, which performs a coherent task of a same kind. 'C' programme does not execute the functions directly, rather it is required to invoke or call such functions. When a function is called in a programme then programme control goes to the function body and it executes the statements, which are involved in a function body. Therefore, it is possible to call function whenever you want to process those function-statements.

In 'C' language, when you call a function you can pass arguments in two ways:

Pass by value

When you use pass-by-value method, the compiler copies the value of an argument in a calling function to a corresponding non-pointer or non-reference parameter in the called function definition. The parameter in the called function is initialised with the value of the passed argument. As long as the parameter has not been declared as constant, the value of the parameter can be changed, but the changes are performed within the scope of the called function only; they have no effect on the value of the argument in the calling function.

In the following example, `main` function passes two values: `5` and `7 to function, func`. The function `func` receives copies of these values and accesses them by the identifiers `a` and `b`. The function `func` changes the value of `a`. When control passes back to `main`, the actual values of `x` and `y` are not changed.

Illustration: The following programme demonstrates calling a function 'by value' method

```
// pass by value
#include < stdio.h>
void func(int a, int b)
{
  printf("Before processing, In func, a = %d    b = %d\n", a, b);
  a += b;
  printf("After processing, In func, a = %d    b = %d\n", a, b);
}
int main()
{
  int x = 5, y = 7;
  printf("Before function call, In main, x = %d   y = %d\n", x,
y);
  func(x, y); //function call by value
  printf("After function call, In main, x = %d    y = %d\n", x,
y);
  return 0;
}
```

Output:

```
Before function call, In main, x = 5    y = 7
```

```
Before processing, In func, a = 5     b = 7
After processing, In func, a = 12     b = 7
After function call, In main, x = 5     y = 7
```

Example 1: Programme to print numbers 1 through 100 without using loop.

```
#include<stdio.h>
int main()
{
    int num = 1;
    print(num);
    return 0;
}
int print(num)
{
    if(num<=100)
    {
        printf("%d ",num);
        print(num+1);
    }
    return(num);
}
```

Output:

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
26 27 28 29 30

 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51
52 53 54 55 56

57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78
79 80 81 82 83

84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100
```

Pass by reference

Passing by reference refers to a method of passing the address of an argument in the calling function to a corresponding parameter in the called function. In 'C', the corresponding parameter in the called function must be declared as a pointer type (an advanced topic, which is discussed later in Module-V; thus, students may better comprehend 'pass by reference' method after reading the pointers in 'C').

In this way, the value of the argument in the calling function can be modified by the called function.

Illustration: The following example shows how arguments are passed by reference. Note that the pointer parameters are initialised with pointer values when the function is called.

```
// pass by reference
#include < stdio.h>
void swapnum(int *i, int *j)
{
  int temp;
  temp = *i;
 *i = *j;
 *j = temp;
}
int main()
{
  int a = 10, b = 20;
  printf("Before function call: a is %d and b is %d\n", a, b);
  swapnum(&a, & b);
  printf("After function call: a is %d and b is %d\n", a, b);
  return 0;
}
```

Note that when the function `swapnum` is called, the actual values of the variables `a` and `b` are swapped because they are passed by reference. This means that using these addresses, we would have an access to the actual arguments. Hence, we would be able to manipulate them. Interestingly, 'C' does not support call by reference, as such! However, it can be simulated using pointers as demonstrated in this illustration.

Output:

```
Before function call: a is 10 and b is 20
After function call: a is 20 and b is 10
```

13.9 Passing Arrays as Arguments

Array arguments are passed in a different way than the single-valued data items. If an array variable name is specified as an actual argument, the individual array elements are not copied. Rather, the location of the array (*i.e.*, the location or address of the first element) is passed to the function. If an element of the array is accessed from within the function, the access will refer to the location of that array element relative to the location of the first element. Thus, any alteration to an array element within the function will carry over to the calling routine. There are other types of data structures that can be passed as arguments to a function. The interested students may like to refer to the resources mentioned under 'Suggested Readings' section at the end of this lesson as well as to the Module V for further exploration in this direction.

Lesson 14

FUNCTIONS IN 'C' – PART II

(Library Functions and Recursion)

14.1 Library Functions

A function, which is predefined in 'C' is known as library function, *e.g.*, `printf`, `scanf`, `getch`, *etc.*, are library functions. Mind it carefully that appropriate header files must be included in the programme for using the library functions. A list of various library functions *vis-à-vis* header files (adopted from Gottfried, 1998) is given in Table-14.1 below.

Example 1: Modified programme (using inbuilt library functions) for Example-2 of Lesson-9 to compute real roots of a quadratic equation (See Example-2 of Lesson-2 for pseudo code and flowchart; and Example-2 of Lesson-9 for simple programme for this problem).

```
#include < stdio.h>
#include < math.h>
int main()
{
   float a,b,c,d, x_1, x_2;
   printf("Enter a, b and c: ");
   scanf("%f %f %f", &a, &b, &c);
   d=sqrt(pow(b,2)-4×a×c); //inbuilt functions used
   x_1=(-b+d)/(2*a);
   x_2=(-b-d)/(2*a);
   printf("a, b, c, x_1, x_2: %5.2f, %5.2f, %5.2f,
          %5.2f,%5.2f",a, b, c, x_1, x_2);
   return 0;
}
```

Test Output:

```
Enter a, b and c: 2 3 1
a, b, c, x_1, x_2:  2.00,  3.00,  1.00, -0.50,-1.00
```

Table 14.1 'C' Library functions *vis-à-vis* header files

Function	Type	Purpose	Header File
`abs(i)`	`int`	Returns the absolute value of `i`.	`stdlib.h`
`acos(d)`	`double`	Returns the arc cosine of `d`.	`math.h`
`asin(d)`	`double`	Returns the arc sine of `d`.	`math.h`

`atan(d)`	`double`	Returns the arc tangent of `d`.	`math.h`
`atan2(d1,d2)`	`double`	Returns the arc tangent of `d1/d2`.	`math.h`
`atof(s)`	`double`	Converts string `s` to a double-precision quantity.	`stdlib.h`
`atoi(s)`	`int`	Converts string `s` to an integer.	`stdlib.h`
`atol(s)`	`long`	Converts string `s` to a long integer.	`stdlib.h`
`calloc(u1,u2)`	`void*`	Allocates memory for an array having `u1`elements, each of length `u2` bytes. Returns a pointer to the beginning of the allocated space.	`malloc.h`, or `stdlib.h`
`ceil(d)`	`double`	Returns a value rounded up to the next higher integer.	`math.h`
`cos(d)`	`double`	Returns the cosine of `d`.	`math.h`
`cosh(d)`	`double`	Returns the hyperbolic cosine of d.	`math.h`
`difftime(l1, l2)`	`double`	Returns the time difference `l1 – l2`, where `l1` and `l2` represent elapsed times beyond a designated base time (see the `time` function).	`time.h`
`exit(u)`	`void`	Closes all files and buffers, and terminate the programme. (Value of `u` is assigned by function, to indicate termination status).	`stdlib.h`
`exp(d)`	`double`	Raises `e` to the power `d` (*e*= 2.7182818… is the base of the natural (Naperian) system of logarithms).	`math.h`
`fabs(d)`	`double`	Returns the absolute value of `d`.	`math.h`
`fclose(f)`	`double`	Closes file `f`. Returns `0` if file is successfully closed.	`stdio.h`
`feof(f)`	`int`	Determines if an *end-of-file* condition has been reached. If so, returns a nonzero value; otherwise, returns `0`.	`stdio.h`
`fgetc(f)`	`int`	Enters a single character from file `f`.	`stdio.h`

fgets(s, i,f)	char*	Enters string s, containing i characters, from file f.	stdio.h
floor(d)	double	Returns a value rounded down to the next lower integer.	math.h
fmod(d1,d2)	double	Returns the remainder of d1/d2 (with same sign as d1).	math.h
fopen(s1,se)	file*	Opens a file named s1 of type s2. Returns a pointer to the file.	stdio.h
fprintf(f,…)	int	Sends data items to file f (remaining arguments are complicated	stdio.h
fputc(c,f)	int	Sends a single character to file f.	stdio.h
fputs(s,f)	int	Sends string s to file f.	stdio.h
fread (s,i1,i2,f)	int	Enters i2 data items, each of size i1 bytes, from file f to string s.	stdio.h
free(p)	void	Frees a block of allocated memory whose beginning is indicated by p.	malloc.h, or stdlib.h
fscanf(f,…)	int	Enters data items from file f (remaining arguments are complicated.	stdio.h
fseek(f,l,i)	int	Moves the pointer for f a distance 1 bytes from location i (i may represent the beginning of the file, the current pointer position, or the end of the file).	stdio.h
ftell(f)	long int	Returns the current pointer position within file f.	stdio.h
fwrite(s, i1,i2,f)	int	Sends i2 data items, each of size i1 bytes, from string s to file f.	stdio.h
getc(f)	int	Enters a single character from file f.	stdio.h
getchar()	int	Enters a single character from the standard input device.	stdio.h
gets(s)	char*		stdio.h

		Enters string s from the standard input device.	
isalnum(c)	int	Determines if argument is alphanumeric. Returns a nonzero value if true; 0 otherwise.	ctype.h
isalpha(c)	int	Determines if argument is alphabetic. Returns a nonzero value if true; 0 otherwise.	ctype.h
isascii (c)	int	Determines if argument is an ASCII character. Returns a nonzero value if true; 0 otherwise.	ctype.h
iscntrl(c)	int	Determines if argument is ASCII control character. Returns a nonzero value if true; 0 otherwise.	ctype.h
isdigit(c)	int	Determines if argument is decimal digit. Returns a nonzero value if true; 0 otherwise.	ctype.h
isgraph(c)	int	Determines if argument is graphic ASCII character (hex 0x21-0x7e; octal 041-176). Returns a nonzero value if true; 0 otherwise.	ctype.h
islower(c)	int	Determines if argument is lowercase. Returns a nonzero value if true; 0 otherwise.	ctype.h
isodigit(c)	int	Determines if argument is an octal digit. Returns a nonzero value if true; 0 otherwise.	ctype.h
isprint(c)	int	Determines if argument is printing ASCII character (hex 0x20-0x7e; octal 040-176). Returns a nonzero value if true; 0 otherwise.	ctype.h
ispunct(c)	int	Determines if argument is a punctuation character. Returns a nonzero value if true; 0 otherwise.	ctype.h

isspace(c)	int	Determines if argument is white space character. Returns a nonzero value if true; 0 otherwise.	ctype.h
isupper(c)	int	Determines if argument is uppercase. Returns a nonzero value if true; 0 otherwise.	ctype.h
isxdigit(c)	int	Determines if argument is a hexadecimal digit. Returns a nonzero value if true; 0 otherwise.	ctype.h
labs(l)	long int	Returns the absolute value of l.	math.h
log(d)	double	Returns the natural logarithm of d.	math.h
log10(d)	double	Returns the logarithm (base 10 of d.)	math.h
malloc(u)	void*	Allocates u bytes of memory. Returns a pointer to the beginning of the allocated space.	malloc.h, or stdlib.h
pow(d1,d2)	double	Returns d1 raised to the d2 power.	math.h
printf(…)	int	Sends data items to the standard output device (arguments are complicated).	stdio.h
putc(c,f)	int	Sends a single character to file f.	stdio.h
putchar(c)	int	Sends a single character to the standard output device.	stdio.h
puts(s)	int	Sends string s to the standard output device.	stdlib.h
rand()	int	Returns a random positive integer.	stdio.h
rewind(f)	void	Moves the pointer to the beginning of file f.	stdio.h
scanf(…)	int	Enters data items from the standard input device (arguments are complicated)	stdio.h
sin(d)	double	Returns the sine of d.	math.h
sinh(d)	double	Returns the hyperbolic sine of d.	math.h

sqrt(d)	double	Returns the square root of d.	math.h
srand(u)	void	Initialises the random number generator.	stdlib.h
strcmp(s1,s2)	int	Compares two strings lexicographically. Returns a negative value if s1<s2; 0 if s1 and s2 are identical; and a positive value if s1>s2.	string.h
strcmpi(s1,s2)	int	Compares two strings lexicographically, without regard to case. Returns a negative value if s1<s2; 0 if s1 and s2 are identical; and a positive value if s1>s2.	string.h
strcpy(s1,s2)	char*	Copies string s2 to string s1.	string.h
strlen(s)	int	Returns the number of characters in a string.	string.h
strset(s,c)	char*	Sets all characters within s to c (excluding the terminating null character \0).	string.h
system(s)	int	Passes command s to the operating system. Returns 0 if the command is successfully executed; otherwise, returns a nonzero value, typically -1.	stdlib.h
tan(d)	double	Returns the tangent of d.	math.h
tanh(d)	double	Returns the hyperbolic tangent of d.	math.h
time(p)	long int	Returns the number of seconds elapsed beyond a designated base time.	time.h
toascii(c)	int	Converts value of argument to ASCII.	ctype.h
tolower(c)	int	Converts letter to lowercase.	ctype.h, or stdlib.h
toupper(c)	int	Converts letter to uppercase.	ctype.h, or stdlib.h

Note: Type refers to data type of the quantity that is returned by the function. An asterisk (*) *denotes a pointer.* c *denotes a character-type argument.* d *denotes a double-precision argument.*

Example 2: Write a function to calculate the real roots of a quadratic equation. Revised version of Example-1 given above.

```
#include <stdio.h>
#include <math.h>
float quad(float a1, float b1, float c1);
main()
{
   float a,b,c;
   printf("Enter a, b and c: ");
   scanf("%f %f %f", &a, &b, &c);
   quad(a, b, c);
   return 0;
}
float quad(float a1, float b1, float c1)
{
   float d, x_1, x_2;
   d=sqrt(pow(b1,2)-4*a1*c1);
   x_1=(-b1+d)/(2*a1);
   x_2=(-b1-d)/(2*a1);
   printf("a, b, c, x_1, x_2: %5.2f, %5.2f, %5.2f,%5.2f, %5.2f",
a1,
            b1, c1, x_1, x_2);
   return 0;
}
```

Test Output:

```
Enter a, b and c: 2 3 1

a, b, c, x_1, x_2:  2.00,  3.00,  1.00,-0.50, -1.00
```

14.2 Recursion

Recursion is a process by which a function calls itself repeatedly, until some specified condition has been satisfied. This process is useful for repetitive computations in which each action is expressed in terms of a previous result. Many iterative problems can be written in this form. In order to solve a problem recursively, two conditions must be satisfied. First, the problem must be written in a recursive form; and second, the problem statement must include a stopping criterion.

```
// Programme to demonstrate function recursion.
```

```
#include <stdio.h>
#include <conio.h>
recursion()
{
     int num;
     printf("\nRecursion... ");
     printf("\n\n Enter Number : ");
     scanf("%d",&num);
     if (num==3)
         {     printf("End of Recursion Function.\n");
          exit(0);}
     else
          recursion();
     return 0;
}
void main()
{
     clrscr();
     recursion();
}
```

Test Output

```
Recursion...
 Enter Number : 4
Recursion...
 Enter Number : 55
Recursion...
 Enter Number : 6543
Recursion...
 Enter Number : 3

End of Recursion Function.
```

Example 3: Programme to multiply two integer numbers by recursion.

```
#include<stdio.h>
int multiply(int x,int y);
int main(){
    int a,b,product;
    printf("Enter any two integers: ");
    scanf("%d%d",&a,&b);
    product = multiply(a,b);
    printf("Multiplication of two integers is %d",product);
```

```
    return 0;
}
int multiply(int x,int y){
    static int product=0,i=0;
    if(i < x){
      product = product + y;
      i++;
      multiply(x,y);
    }
    return product;
}
```

Test Output:

```
Enter any two integers: 5 8
Multiplication of two integers is 40.
```

Example 4: Programme to find the sum of first n natural numbers by recursion:

```
#include<stdio.h>
int main(){
    int n,sum;
    printf("Enter the value of n: ");
    scanf("%d",&n);
    sum = getSum(n);
    printf("Sum of n numbers: %d",sum);
    return 0;
}
int getSum(n){
    static int sum=0;
    if(n>0){
         sum = sum + n;
         getSum(n-1);
    }
    return sum;
}
```

Test output:

```
Enter the value of n: 10
Sum of n numbers: 55
```

Note that recursion can be used as an alternative to looping structures.

Example 5: Programme to compute factorial of a positive integer number.

Commonly, this problem is defined as $n! = 1 \times 2 \times 3 \times \ldots \times n$, where n is the given integer number. However, you can express this problem in another way, by writing $n! = n \times (n - 1)$ This is a recursive statement of the problem, in which the desired action is expressed in terms of previous result. Also, you know that $1! = 1$ by definition, therefore, this provides the stopping criterion. The following programme accomplishes this task.

```
#include <stdio.h>

     long int factorial(int n);
     void main()
     {
          int n;
          long int factorial(int n);
          printf("n = ");
          scanf("%d", &n);
          printf("n! = %ld\n", factorial(n));
     }
     long int factorial(int n)
     {
          if (n <= 1)
               return(1);
          else
               return(n * factorial(n-1));
     }
```

Test Output

```
n = 10
n! = 3628800
```

Explanation:

The `main` portion of the programme simply reads the integer quantity *n* and then references the long-integer recursive function, `factorial`. Note that the long integers are used for this calculation as factorials of even modest values of *n* are large integer quantities. The function, `factorial` accesses itself recursively, with an actual argument *(n-1)* that decreases in magnitude for each successive access. The recursive access terminates when the value of the actual argument becomes equal to 1. Note that when the programme is executed, the function, `factorial` will be accessed repeatedly, once in `main` and *(n-1)* times within itself, though the end-user using this programme will not be aware of this. Only resultant value will be displayed. For example, for the value of `n=10`, the factorial is `3628800`. The input and output of the programme is:

```
n = 10
```

```
n! = 3628800
```

Lesson 15

ARRAYS in 'C' – PART I

(Single-Dimensional Arrays: Declaring and Assigning Initial Values)

15.1 Arrays

Arrays are the data structure, which hold multiple variables of the same data type; and each element of the set can then be referenced by means of a number called index number or subscript. In array elements, index or subscript begins with number zero. An array variable name like other variables in 'C' must be declared before it is used. In other words, an array in 'C' can be defined as a number of memory locations each of which can store the same data type and that can be referenced through the same variable name. In array elements, index or subscript begins with number zero and takes values up to 1 less than maximum size of the array. The general syntax:

```
type variable-name[Size];
```

The keyword 'type' specifies the data type of the elements that will be contained in the array, such as int, float or char and the Size (enclosed between square braces) indicates the maximum number of elements that can be stored inside the array, *i.e.*, from '0' to (*Size-1*). For instance, consider the following statement:

```
float height[50];
```

This example declares the variable, height to be an array containing 50 elements of floating-point data type. Any subscripts within the range 0 to 49 are valid, *e.g.*, height[0] refers to the first element and height[49] refers to the last element of the array.

15.2 Declaring Arrays

Arrays may consist of any of the valid data types. Arrays are declared along with the other variables in the declaration section of the programme.

An illustration

```
int numbers[100];
numbers[2]=10;
--numbers[2];
printf("The third element of the array, numbers is
%d\n", numbers[2]);
```

Output:

```
The third element of the array, numbers is 9.
```

The above programme segment declares an array, `numbers` of size `100` elements; assigns the value of `10` to the third element of the array, `numbers;` followed by decrementing this value by one; and finally prints the value of the third element of the array, `numbers`.

15.3 Assigning Initial Values to Arrays

External and static variables can be initialised if so desired. The initial values are enclosed in curly brackets and must appear in the same order as they will be assigned elements. The general syntax is:

```
type array _name[size] = {element_1, element_2, …,
                          element_n}
```

An illustration

```
int x;
static int values[] = {1,2,3,4,5,6,7,8,9};
for(x = 0; x < 9; ++x)
    printf("Values [%d] = %d\n", x, values[x]);
```

Output:

```
Values [0] = 1
Values [1] = 2
.
.
.
Values [8] = 9.
```

This programme declares an array, `values`. Note that inside the square brackets, there is no variable to indicate as to how big the array will be. In this case, 'C' initialises the array to the number of elements that appear within the square braces. For instance, the array, `values` consists of nine elements (`0` to `8`).

Example 1:

Programme to demonstrate one dimensional array. It accepts three numbers to be entered through the keyboard at run-time; stores these numbers in a one-dimensional array `a`; and finally, prints the array `a`.

```
#include < stdio.h>
#include <conio.h>
void main()
{
      int a[3], i;;
      clrscr();
      printf("\n\t Enter three numbers : ");
```

```
        for(i=0; i<3; i++)
        {
              scanf("%d", &a[i]);  // read array
        }
        printf("\n\n\t Numbers are : ");
        for(i=0; i<3; i++)
        {
              printf("\t %d", a[i]);  // print array
        }
        getch();
}
```

Test Output

```
Enter three numbers : 9 4 6
Numbers are :     9     4     6
```

Example 2:

Programme to sort a given array of numbers into ascending order using insertion sort method (See Example-6 of Lesson-2 for pseudo code).

```
#include<stdio.h>
#include<conio.h>
  void insertion(int a[],int n)
   {
    int i,j,x,k;
    for(i=1;i<=n-1;i++)
    {
       j=i;
       x=a[i];
       while(a[j-1]>x && j>0)
       {
           a[j]=a[j-1];
           j=j-1;
       }
       a[j]=x;
       printf("\n\n The array after pass no.%d: ",i);
       for(k=0;k<=n-1;k++)
          printf("%4d",a[k]);
    }//end for.
  } //end function.
  void main()
   {
```

```
    int a[1000],n,i;
    clrscr();
    printf("\n\nEnter an integer value for total nos. of
            elements to be sorted: ");
    scanf("%3d",&n);
    for(i=0;i<=n-1;i++)
    {
      printf("\n\nEnter an integer value for element
             no.%d:",i+1);
      scanf("%4d",&a[i]);
    }
    insertion(a,n);
    printf("\n\n\nFinally sorted array is : ");
    for(i=0;i<=n-1;i++)
       printf("%4d",a[i]);
  }//end programme.
```

Test Output:

```
Enter an integer value for total no.s of elements to be sorted: 7

Enter an integer value for element no.1: 12

Enter an integer value for element no.2: 9

Enter an integer value for element no.3: 3

Enter an integer value for element no.4: 4

Enter an integer value for element no.5: 22

Enter an integer value for element no.6: 45

Enter an integer value for element no.7: 12

 The array after pass no.1:    9  12   3   4  22  45  12

 The array after pass no.2:    3   9  12   4  22  45  12

 The array after pass no.3:    3   4   9  12  22  45  12

 The array after pass no.4:    3   4   9  12  22  45  12

 The array after pass no.5:    3   4   9  12  22  45  12

 The array after pass no.6:    3   4   9  12  12  22  45

 Finally sorted array is :    3   4   9  12  12  22  45
```

Lesson 16

ARRAYS in 'C' – PART II
(Multi-Dimensional Arrays and Pointers)

16.1 Multi-dimensional Arrays

'C' allows arrays to have dimensions more than two. Multi-dimensional arrays have two or more index values, which specify the elements in the array. The general syntax is:

```
type variable-name[i][j]
```

The `type` specifies the data type of the elements that will be contained in the array, such as `int`, `float` or `char` and the first index value `i` specifies the row index, while `j` specifies the column index of the variable.

Illustration

```
int matrix1[3][3];
```

Here, `matrix1` is a two-dimensional array of the type, integer having three rows and three columns; and, hence, can store up to nine (3×3) elements. The `matrix1` can be initialised by assigning value to each element as follows:

```
int matrix[3][3]={{2,4,7}, {4,9,0},{3,9,6}};
```

It will appear as shown below:

```
matrix1[0][0]=2
matrix1[0][1]=4
matrix1[0][2]=7
matrix1[1][0]=4
matrix1[1][1]=9
matrix1[1][2]=0
matrix1[2][0]=3
matrix1[2][1]=9
matrix1[2][2]=6
```

Example 1: Programme to multiply two 3 × 3 matrices.

```
#include <stdio.h>
#include <conio.h>
void main()
{
 int a[3][3], b[3][3], c[3][3], i, j, k;
 clrscr();
 printf("Enter the elements of first 3x3 matrix");
 for (i = 0;i < 3; i++)
```

```
{
  for (j = 0; j < 3; j++)
  {
     scanf("%d",&a[i][j]);
  }
}
printf("\nEnter the elements of second 3×3 matrix");
for(i = 0; i < 3; i++)
{
  for (j = 0; j < 3; j++)
  {
     scanf("%d", &b[i][j]);
  }
}
printf("\nThe first matrix is :-\n");
for (i = 0; i < 3; i++)
{
  for (j = 0; j < 3; j++)
  {
     printf("\t%d", a[i][j]);
  }
  printf("\n");
}
printf("\nThe second matrix is :-\n");
for (i = 0; i < 3; i++)
{
  for (j = 0; j < 3; j++)
  {
     printf("\t%d", b[i][j]);
  }
  printf("\n");
}
printf("\nMultiplication of the two matrices is as
        follows:\n");
for (i = 0;i < 3; i++)
{
  printf("\n");
  for (j = 0; j < 3; j++)
  {
    c[i][j]=0;
```

```
      for(k=0;k<3;k++)
        c[i][j] = c[i][j]+a[i][k] * b[k][j];
      printf("\t%d", c[i][j]);
    }
  }
  getch();
}
```

Test Output

```
Enter the elements of first 3x3 matrix1
2
3
4
5
6
7
8
9

Enter the elements of second 3x3 matrix1
2
3
4
5
6
7
8
9

The first matrix is :-
        1       2       3
        4       5       6
        7       8       9

The second matrix is :-
        1       2       3
        4       5       6
        7       8       9

Multiplication of the two matrices is as follows:
        30      36      42
        66      81      96
```

```
        102     126     150
```

Example 2: Programme to transpose a 3 × 3 matrix.

```
#include < stdio.h >
#include < conio.h >
void main()
{
   int arr[3][3],i,j;
   clrscr();
   printf("Enter elements for the array \n");
   for(i=0;i < 3;i++)
   {
     for(j=0;j < 3;j++)
     {
         scanf("%d",&arr[i][j]);
     }
   }
   printf("Original array entered by the user is \n");
   for(i=0;i < 3;i++)
   {
     for(j=0;j < 3;j++)
     {
       printf("%d ",arr[i][j]);
     }
     printf("\n");
   }
   printf("\n Transpose of the array is \n");
   for(i=0;i < 3;i++)
   {
      for(j=0;j < 3;j++)
        printf("%d ",arr[j][i]);
      printf("\n");
   }
   getch();
}
```

Test Output:

```
Enter elements for the array
5
6
7
```

```
3
5
4
1
7
8

Original array entered by the user is

5 6 7
3 5 4
1 7 8

 Transpose of the array is

5 3 1
6 5 7
7 4 8
```

Example 3: Programme to compute sum of diagonal elements of a matrix.

```
#include<stdio.h>
int main()
{
  int a[10][10],i,j,sum=0,m,n;
  printf("\nEnter the row and column of matrix: ");
  scanf("%d %d",&m,&n);
  printf("\nEnter the elements of matrix: ");
  for(i=0;i<m;i++)
      for(j=0;j<n;j++)
           scanf("%d",&a[i][j]);
  printf("\nThe matrix is\n");
  for(i=0;i<m;i++){
      printf("\n");
      for(j=0;j<m;j++){
      printf("%d\t",a[i][j]);
      }
 }
 for(i=0;i<m;i++){
     for(j=0;j<n;j++){
          if(i==j)
              sum=sum+a[i][j];
     }
 }
```

```
 printf("\n\nSum of the diagonal elements of the matrix
        is: %d",sum);
 return 0;
}
```

Test Output

```
Enter the row and column of matrix: 3 3
Enter the elements of matrix: 2
3
5
6
7
9
2
6
7

The matrix is

2       3       5
6       7       9
2       6       7
Sum of the diagonal elements of the matrix is: 16
```

Example 4: Programme to find determinant of a 3 × 3 matrix.

```
#include<stdio.h>
int main()
{
  int a[3][3],i,j;
  long determinant;
  printf("Enter the 9 elements of matrix: ");
  for(i=0;i<3;i++)
      for(j=0;j<3;j++)
           scanf("%d",&a[i][j]);
  printf("\nThe matrix is\n");
  for(i=0;i<3;i++){
      printf("\n");
      for(j=0;j<3;j++)
           printf("%d\t",a[i][j]);
  }
```

```
  determinant = a[0][0]*((a[1][1]*a[2][2]) - (a[2][1]*a[1][2]))
-a[0][1]*(a[1][0]*a[2][2] - a[2][0]*a[1][2]) + a[0][2]*(a[1][0]*a
[2][1] - a[2][0]*a[1][1]);
printf("\nDeterminant of 3×3 matrix: %ld",determinant);
  return 0;
}
```

Test Output

```
Enter the 9 elements of matrix: 1
2
3
4
5
6
7
8
9

The matrix is

1       2       3
4       5       6
7       8       9

Determinant of 3×3 matrix: 0
```

Example 5: Programme to find inverse of a 3×3 matrix.

```
#include<stdio.h>
int main()
{
  int a[3][3],i,j;
  float determinant=0;
  printf("Enter the 9 elements of matrix: ");
  for(i=0;i<3;i++)
      for(j=0;j<3;j++)
           scanf("%d",&a[i][j]);
  printf("\nThe matrix is\n");
  for(i=0;i<3;i++){
      printf("\n");
      for(j=0;j<3;j++)
           printf("%d\t",a[i][j]);
  }
  for(i=0;i<3;i++)
```

```
        determinant = determinant + (a[0][i]*(a[1][(i+1)%3]*
                      a[2][(i+2)%3] - a[1][(i+2)%3]
                      *a[2][(i+1)%3]));
  printf("\nInverse of matrix is: \n\n");
  for(i=0;i<3;i++){
      for(j=0;j<3;j++)
           printf("%.2f\t",((a[(i+1)%3][(j+1)%3] *
                  a[(i+2)%3][(j+2)%3]) -
                  (a[(i+1)%3][(j+2)%3]*a[(i+2)%3][(j+1)%3]))/
                  determinant);
           printf("\n");
   }
   return 0;
}
```

Test Output:

```
Enter the 9 elements of matrix: 3
5
2
1
5
8
3
9
2
The matrix is
3       5       2
1       5       8
3       9       2
Inverse of matrix is:
0.70    -0.25   0.07
-0.09   -0.00   0.14
-0.34   0.25    -0.11
```

Illustration:

Consider the following code segment demonstrating initialisation of an array with characters to form a string and also, some important tips to properly handle the strings in ‘C’.

```
static char name1[] = {'H','e','l','l','o'};
static char name2[] = "Hello";
```

```
printf("%s\n", name1);
printf("%s\n", name2);
```

Output *(the typical output shown below was produced as a result of executing the above code using the Turbo C++ Version 3.0 compiler by Borland International, Inc.)*:

```
HelloHello
Hello
```

Note that the difference between the two arrays `name1` and `name2` is that the latter has a null value placed automatically at the end of the string, during the initialisation process, *i.e.*, null value is stored at the location, `name2[5]`; however, the array, `name1` does not get a null character placed after initialisation, thereby often leading to some garbage (superfluous) characters being printed at the end, *e.g.*, in this example, `HelloHello` is printed instead of `Hello`. This can be remedied by inserting a null character at the end of the array, `name1` as follows:

```
static char name1[] = {'H','e','l','l','o','\0'};
```

Revised output (the revised output shown below was produced as a result of executing the modified code as discussed above, which is the desired and proper result):

```
Hello
Hello.
```

16.2 Pointers

A pointer is a variable suitable for keeping memory addresses of other variables; the values assigned to a pointer are memory addresses of other variables or other pointers. 'C' pointers are characterised by their value and data type. The value is the address of the memory location the pointer points to, the type determines as to how the pointer will be incremented and/or decremented in pointer or subscript arithmetic.

Pointers are used to manipulate arrays and they can be used to return more than one value from a function. Pointers are declared by using the asterisk '*', *e.g.*, see the following statement showing a pointer:

```
int *p;
```

Each variable has two attributes: address and value. The address is the location in memory. In that location, the value is stored. During the lifetime of the variable, the address is not changed; however, the value may change.

```
#include
void main (void)
{
  int i;
  int * a;
  i = 10;
```

```
    a = &i;
    printf (" The address of i is %8u \n", a);
    printf (" The value at that location is %d\n", i);
    printf (" The value at that location is %d\n", *a);
}
```

Output (the typical output shown below was produced as a result of executing the above code using the Turbo C++ Version 3.0 compiler by Borland International, Inc.):

```
The address of i is   65524

The value at that location is 10

The value at that location is 10
```

Lesson 17

STRUCTURES AND UNIONS

17.1 Structures

Recall that the arrays are used to store a large set of data as well as to manipulate these data. However, there is a limitation with the arrays that all the elements stored in an array are to be of the same data type! If you need to use a collection of items with different data types, it is not possible using an array. Instead, a `structure` is used.

A `structure` can be considered as a template used for defining a collection of variables under a single name. Structures facilitate a programmer to group several elements of different data types into a single logical unit. Thus, a `structure` is a collection of one or more variables, possibly of different data types, grouped together under a single name for convenient handling.

The general syntax of `structure` is:

```
struct tag_name
{
      data type member1;
      data type member2;
      . . .
};
```

For example, let us store a date inside a 'C' programme. This can be defined as a `structure` called `date` with three elements: `day`, `month` and `year` as follows:

```
struct date
{
      int day;
      int month;
      int year;
};
```

An illustration

The following code segment demonstrates the structure named `lib_book` (*i.e.*, a collection of library books), which includes members as `title` (book title), `author1` (first author), `author2` (second author), `ac_no` (book accession number), `pub_name` (publisher's name), `pages` (number of pages of the book) and `price` (unit price of the book in rupees). Note that the data types of the `structure` members are different unlike an array variable.

```
struct lib_books
{
      char title[20];
```

```
        char author1[15];
        char author2[15];
        char ac_no[10];
        char pub_name[20];
        int pages;
        float price;
};
```

The keyword `struct` declares a structure to hold the details of six fields as described above. The tag name (`lib_books`) is used to define a data object, 'library books' comprising of several members. Hence, the above defined structure is not a variable but a template for the object. 'C' allows a programmer to store both simple as well as sophisticated data types within an array, *i.e.*, a programmer-defined `structure` can be an element of an array.

Generally, a `structure` is defined at the beginning of a programme, *i.e.*, just following the `main()` statement. However, the `structure` definition statement merely instructs the compiler that a `structure` exists, but it does not assign any memory to the variable. The memory allocation takes place only when a `structure` variable is declared. We can declare `structure` variables using the tag name anywhere in the programme. For example, the statement:

```
struct lib_books book1,book2,book3;
```

declares `book1`, `book2` and `book3` as variables of the type `struct lib_books`; each declaration has four elements of the structure `lib_books`. The complete structure declaration looks like:

```
struct lib_books
{
        char title[20];
        char author1[15];
        char author2[15];
        char ac_no[10];
        char pub_name[20];
        int pages;
        float price;
};
struct lib_books, book1, book2, book3;
```

A structure does not occupy any memory until it is associated with the structure variable such as `book1`. The template is terminated with a semicolon. While the entire declaration is considered as a statement, each `member` is declared independently for its `name` and `type` in a separate statement inside the template. The tag name such as `lib_books` is used to declare structure variables of this data type later in the programme.

Also, both the template declaration as well as the variables' declaration can be combined in a single statement as follows:

```
struct lib_books
{
      char title[20];
      char author1[15];
      char author2[15];
      char ac_no[10];
      char pub_name[20];
      int pages;
      float price;
} book1, book2, book3;
```

The use of tag name is optional, *e.g.*, see the following code segment:

```
struct
{
      char title[20];
      char author1[15];
      char author2[15];
      char ac_no[10];
      char pub_name[20];
      int pages;
      float price;
} book1, book2, book3;
```

The variables, `book1`, `book2` and `book3` represent structure variables signifying three books, but the declaration statement does not include a tag name! A structure is usually defined prior to `main` function along with macro definitions. In such cases, the structure assumes global status and all the functions can access the structure.

17.2 Giving Values to the Members

As stated earlier, the members themselves are not variables, however, they should be linked to `structure` variables so as to make them meaningful members. The link between a member and a variable is established using the member operator, '.', which is known as dot operator or period operator, *e.g.*, `book1.price` is the variable representing the `price` of `book1` and can be treated like any other ordinary variable. Let us make use of the `scanf` statement to assign values as described below:

```
scanf("%s", book1.file);
scanf("%d",&book1.pages);
```

Alternatively, you can assign variables to the members of `book1` as follows:

```
strcpy(book1.title, "basic");
strcpy(book1.author, "A. K. Sharma");
book1.pages=187;
book1.price=200.00;
```

17.3 Initialising `structure`

Like other data types, you can initialise `structure` while declaring the same. The `structure` construct obeys the same set of rules as arrays. Thus, the fields of a `structure` are initialised by declaring the `structure` with a list containing values for each field as for arrays. This is demonstrated with the help of following examples.

Example 1: The following code demonstrates the use of `structure` construct including the `structure` definition, inputting and storing data into the memory and retrieving information from the memory.

```
#include <stdio.h>
void main()
{
     struct students
     {
          int id;
          char name[20];
          char address[20];
          char stream[3];
          int age;
     } newstudents;
     printf("Input the Students Information:\n\n");
     printf("The student identification number  ");
     scanf(" %d",&newstudents.id);
     printf("The name of the student  ");
     scanf(" %s",&newstudents.name);
     printf("The mailing address of the student  ");
     scanf(" %s",&newstudents.address);
     printf("The stream to which the student belongs  ");
     scanf(" %s",&newstudents.stream);
     printf("The age (in years) of the student  ");
     scanf(" %d",&newstudents.age);
     printf("\n\n");
     printf("Output Students Information:  \n\n");
     printf("Student Identity Number = %d\n", newstudents.id);
     printf("Student Name = %s\n", newstudents.name);
     printf("Student Address = %s\n", newstudents.address);
```

```
    printf("Student Stream = %s\n", newstudents.stream);
    printf("Age (in years) of Student = %d\n", newstudents.age);
}
```

The input data fed through the keyboard and the output produced (verbatim) when the above code was executed, are given below:

```
Input the Students Information:

The student identification number  009
The name of the student  Aashish
The mailing address of the student  8,Def.Col.,Ambala
The stream to which the student belongs  E
The age (in years) of the student  19

Output Students Information:

Student Identity Number = 9
Student Name = Aashish
Student Address = 8,Def.Col.,Ambala
Student Stream = E
Age (in years) of Student = 19
```

17.4 Structures Containing Arrays

There can also be structures containing arrays as its elements. This is illustrated with the help of the following example.

Example 2: The following code demonstrates the use of '`structure`' construct including the `structure` definition, initialisation the `structure` with data and retrieving information from the `structure`.

```
#include <stdio.h>
void main()
{
struct students
{
    int id;
    char name[20];
    char address[20];
    char stream[3];
    int age;
};
struct students newstudents = {9, "Aashish", "HEC College","CE",
19};
printf("\n\n");
```

```
printf("Output Students Information:  \n\n");
printf("Student Identity Number = %d\n", newstudents.id);
printf("Student Name = %s\n", newstudents.name);
printf("Student Address = %s\n", newstudents.address);
printf("Student Stream = %s\n", newstudents.stream);
printf("Age (in years) of Student = %d\n", newstudents.age);
}
```

```
Output Students Information:

Student Identity Number = 9
Student Name = Aashish
Student Address = HEC College
Student Stream = CE
Age (in years) of Student = 19
```

As stated earlier, an array can contain a structure as one of its elements. Referencing an element in the array containing a structure as an element is quite simple. Also, initialisation of structure arrays is similar to initialisation of multidimensional arrays. Do not confuse this concept with array of structures.

17.5 Arrays of Structure

It is possible to define an array of structures, for example, if you are maintaining information about students of a college and there are 100 students studying in the college. You need to use an array rather than simple variables. You can define an array of structures as shown in following code that demonstrates the use of '`structure`' construct including the `structure` definition, initialisation, the `structure` with multiple data records and retrieving information from the `structure`:

```
// Code demonstrating use of a structure
#include <stdio.h>
void main()
{
     struct students
     {
          int id;
          char name[20];
          char address[20];
          char stream[3];
          int age;
     };
     struct students newstudents[] = {
                    {9, "Aashish", "HEC College", "CSE", 19},
```

```
                    {10,"Anuj","HEC College","EE",19},
                    {11, "Manpreet", "SPCET", "CE",19}
                                    };
    printf("\n\n");
    printf("Output Students Information:  \n\n");
    for (int i=0; i<3; i++)
    {
        printf("Student Identity Number = %d\n", newstudents
        [i].id);
        printf("Student Name = %s\n", newstudents[i].name);
        printf("Student Address = %s\n", newstudents
        [i].address);
        printf("Student Stream = %s\n", newstudents[i].stream);
        printf("Age (in years) of Student = %d\n\n",
        newstudents[i].age);
    }
}
```

```
Output Students Information:

Student Identity Number = 9
Student Name = Aashish
Student Address = PEC College
Student Stream = CSE‼
Age (in years) of Student = 19

Student Identity Number = 10
Student Name = Anuj
Student Address = HEC College
Student Stream = EE
Age (in years) of Student = 19

Student Identity Number = 11
Student Name = Manpreet
Student Address = SPCET
Student Stream = CE
Age (in years) of Student = 19
```

17.6 Structures and Functions

A `structure` can be passed as an argument to functions. Unlike array names, which always point to the start of the array, `structure` names are not pointers. Thus, if a change is made to the structure parameter(s) inside a function, the corresponding arguments remain unaffected.

Passing structure to elements to functions

A `structure` may be passed into a function as individual member or a separate variable. The following programme illustrates as to how to display the contents of a `structure` by passing its individual elements to a function.

```
# include <stdio.h>
display(int e_no, char e_name[25], float e_sal);
void main()
{
  struct employee
  {
     int emp_id;
     char name[25];
     char dept[10];
     float salary;
  };
  struct employee emp1 = {267,"Aashish","CSE", 50000.00};
  display(emp1.emp_id, emp1.name, emp1.salary);
}

// function to display structure variables

display(int e_no, char e_name[25], float e_sal)
  {
    printf(" %d %s %f ", e_no, e_name, e_sal);
    return 0;
  }
```

The programme output is given below:

Output:

```
267 Aashish 50000.000000
```

In this example, the structure members, `emp_id`, `name` and `salary` have been declared as integer type variable, character type array variable and floating-point variable, respectively.

Now, the `display` function is called using statement,

```
display(emp1.emp_id,emp1.name,emp1.salary);
```

i.e., the structure members, `emp_id`, `name` and `salary` are transmitted to the `display` function. Evidently, passing individual members would become more tedious as the number of

structure elements go on increasing; hence, a better way would be to pass the entire structure at a time. This is illustrated with the help of the following programme:

```
#include < stdio.h>
display(struct employee empf);
struct employee
{
     int emp_id;
     char name[25];
     char dept[10];
     float salary;
};
void main()
{
     struct employee emp1={12,"Aashish","Computer",75000.00};
     //sending entire employee structure
     display(emp1);
}
//function to pass entire structure variable
display(struct employee empf)
{
     printf("%d  %s  %s  %f",  empf.emp_id,  empf.name,  empf.dept,
     empf.salary);
     return 0;
}
12 Sadanand Computer 7500.000000
```

17.7 Unions

A union is a variable, which may hold members of different sizes and types. The union, like structure, contains members whose individual data types may differ from each other. However, the members that compose a union all share the same storage area within the computer's memory whereas each member within a structure is assigned its own unique storage area. Thus, union is used to conserve memory. It is useful for applications involving multiple members where values are not assigned to all the members at any one point of time. Like structure, union can be declared using the keyword union as follows:

```
union item
{
     int m;
     float p;
     char c;
} code;
```

This declares a variable, code of the type union item. The union contains three members each with a different data type. However, only one of them can be used at a time. This is because only one location is allocated for union variable irrespective of size. The compiler allocates a piece of storage that is large enough to access a union member; the same syntax is used to access union members as structure members. That is, code.m, code.p and code.c are all valid member variables. While accessing such member variables, it should be ensured that an access is being made to the member whose value is currently stored. For example a statement such as

```
code.m=456;
code.p=456.78;
Printf("%d",code.m);
```

would produce erroneous result.

Actually, a union creates a storage location that can be used by one of its members at a time. When a different number is assigned a new value, the new value supersedes the previous member's value. The union may be used in any situations where an application of the structure is possible.

```
# include <stdio.h>
main( ){
union
{
    int one;
    char two;
} val;
val.one = 300;
printf("val.one = %d \n", val.one);
printf("val.two = %d \n", val.two);
}
```

In the above code, there are two members: int one and char two. The member one is initialised to 300. Note that only one member the union has been initialised; however, using two different printf statements, the individual members of the union val are displayed as:

```
val. one = 300
val. two = 44
```

Since the char variable two was not initialised, the second printf statement produces a random value of 44.

While using a union, the programmer should keep track of whatever type is put into it and retrieve the right type at the right time. Here's another example:

```
#include <stdio.h>
```

```
#include <stdlib.h>
main(){
        union {
                float u_f;
                int u_i;
        }var;
        var.u_f = 23.5;
        printf("value is %f\n", var.u_f);
        var.u_i = 5;
        printf("value is %d\n", var.u_i);
        exit(EXIT_SUCCESS);
}
```

Output

```
value is 23.500000
value is 5
```

Lesson 18

FILE HANDLING IN 'C'

18.1 Introduction

Real life situations involve large volumes of data. In such cases, the console (keyboard) oriented I/O operations create two major problems: *a)* it becomes cumbersome and time consuming to handle large volumes of data through terminals; and *b)* the entire data is lost when either the programme is terminated or computer is turned off. Thus, it is necessary to have more flexible approach where data can be stored on the disks and read whenever necessary, without destroying the data. This method employs the concept of files to store data.

In this section, you will learn about files, which are very important for storing information permanently. You store information in files for many purposes like data processing by your programmes.

What is a File?

A file is a collection of bytes stored on a secondary storage device, which is generally a disk of some kind. The collection of bytes may be interpreted, for example, as characters, words, lines, paragraphs and pages from a textual document; fields and records belonging to a database; or pixels from a graphical image. The meaning attached to a particular file is determined entirely by the data structures and operations used by a programme to process the file. A file is simply a machine decipherable storage medium where programmes and data are stored for machine usage. Essentially, there are two kinds of file that programmers deal with text files and binary files.

ASCII Text Files

A text file can be a stream of characters that a computer can process sequentially. It is not only processed sequentially but only in forward direction. For this reason a text file is usually opened for only one kind of operation (reading, writing, or appending) at any given time.

Similarly, since text files only process characters, they can only read or write data one character at a time. However, 'C' provides several functions that deal with lines of text, but these still essentially process data one character at a time. A text stream in 'C' is a special kind of file. Depending on the requirements of the operating system, newline characters may be converted to or from carriage-return/linefeed combinations depending on whether data is being written to, or read from, the file. Other character conversions may also occur to satisfy the storage requirements of the operating system. These translations occur transparently and they occur because the programmer has signalled the intention to process a text file.

Binary Files

A binary file is similar to a text file. It is a collection of bytes. In 'C', a byte and a character are equivalent. Hence, a binary file is also referred to as a character stream, but there are two essential differences.

1. No special processing of the data occurs and each byte of data is transferred to or from the disk unprocessed.
2. 'C' places no constructs on the file, and it may be read from, or written to, in any manner chosen by the programmer.

Binary files can be either processed sequentially or, depending on the needs of the application, they can be processed using random access techniques. In 'C', processing a file using random access techniques involves moving the current file position to an appropriate place in the file before reading or writing data. This indicates a second characteristic of binary files. They a generally processed using read and write operations simultaneously. For example, a database file will be created and processed as a binary file. A record update operation will involve locating the appropriate record, reading the record into memory, modifying it in some way, and finally writing the record back to disk at its appropriate location in the file. Such operations are common to many binary files, but are rarely found in applications that process text files.

18.2 Reading and Writing Files

This section describes as to how to read from and write onto files in 'C'. All file functions need `stdio.h` header file to work properly. The first thing you need to know about is file pointers. File pointers are like any other pointer, but they point to a file. You define a file pointer as follows:

```
FILE *filepointer;
```

The type FILE is used for a file variable and is defined in the `stdio.h` file. It is used to define a file pointer for use in file operations. A list of file operation functions along with their purposes is given in Table-18.1 below.

Table 18.1 File operation functions

Function Name	Operation
`fopen`	Creates a new file. Opens an existing file.

`fclose`	Closes a file which has been opened for use
`getc`	Reads a character from a file
`putc`	Writes a character to a file
`fprintf`	Writes a set of data values to a file
`fscanf`	Reads a set of data values from a file
`getw`	Reads a integer from a file
`putw`	Writes an integer to the file
`fseek`	Sets the position to a desired point in the file
`ftell`	Gives the current position in the file
`rewind`	Sets the position to the beginning of the file

Before you write to a file, you must open it. This indicates to the system that you want to write to a file and you also define its file name with the `fopen` function illustrated below. The file pointer, *e.g.*, `filepointer` in the following example, points to the file and two arguments are required in the parentheses, *i.e.*, the file name, followed by the file type as follows:

```
filepointer = fopen("filename", "mode");
```

`fopen` returns a file pointer. It returns `NULL` if the file does not exist. `fopen` takes the first argument as the `filename` to open. It needs to be of string type. The second argument is the mode argument.Mode specifies what you want to do with the file. Some modes are: `"r"` - read the file; `"w"` - write the file; `"a"` - append to the file; `"r+"` - read and write to the file; `"w+"` - read and write, overwrite the file; and `"a+"` - read and write, append. These modes will open files in text mode. Files opened in text mode have some bytes filtered out. If you want to open binary files, use binary mode by adding a `"b"` to the mode. For example: `"rb"` - read the file in binary mode. To read a character from a file, you use `fgetc`. It is like `getchar`, but for files. It works like this:

```
character = fgetc(filepointer);
```

`fgetc` returns the character that is read from the file as an integer. `fgetc` takes the file pointer as its only input. It will automatically increment the pointer to read the next character. `fputc` allows you to write a character to a file:

```
fputc(character, filepointer);
```

`fputc` takes an `unsigned char` as the first argument and the file pointer as the second argument. `fputc` returns `EOF` on failure. You can also use `fprintf` and `fscanf`. They work like `printf` and `scanf`, except the file pointer is the first argument. They work like this:

```
//writes hello world to the file
fprintf(filepointer, "Hello, World!\n");
//reads an integer from the file
fscanf(filepointer, "%d", integer);
```

In order to close the file again, you must use `fclose`. It looks like this:

```
fclose(filepointer);
```

`fclose` closes the file that filepointer points to.

Example 1: Programme to read a file.

The following programme reads a file entered by the user and displays its contents on the screen, `fopen` function is used to open a file; it returns a pointer to structure `FILE`. `FILE` is a predefined structure in `stdio.h`. If the file is successfully opened then `fopen` returns a pointer to file; otherwise if it is unable to open a file then it returns NULL value. `fgetc` function returns a character, which is read from the file and `fclose` function closes the file.

Opening a file means bringing file from secondary storage (hard disk) to RAM so as to perform operations on it. The file must be present in the directory in which the executable file of this code is present. The code is as follows:

```
#include<stdio.h>
#include<stdlib.h>
main()
{
   char ch, file_name[25];
   FILE *fp;
   printf("Enter the name of file you wish to see ");
   gets(file_name);
   fp = fopen(file_name,"r"); // read mode
   if( fp == NULL )
   {
      perror("Error while opening the file.\n");
      exit(EXIT_FAILURE);
   }
   printf("The contents of %s file are :- \n\n", file_name);
   while( ( ch = fgetc(fp) ) != EOF ) //EOF denotes End-of-File
         printf("%c",ch);
   fclose(fp);
   return 0;
}
```

Example 2: Programme to copy a file.

This programme copies a file. Firstly, you will specify the file name with proper path, to copy and then you will enter the name of target file. Also, you must specify the extension of the file. The source file is opened in read mode (`"r"`) and target file in write mode (`"w"`).

```
#include <stdio.h>
```

```
#include <stdlib.h>
main()
{
   char ch, source_file[20], target_file[20];
   FILE *source, *target;
   printf("Enter name of file to copy\n");
   gets(source_file);
   source = fopen(source_file, "r");
   if( source == NULL )
   {
      printf("Press any key to exit...\n");
      exit(EXIT_FAILURE);
   }
   printf("Enter name of target file\n");
   gets(target_file);
   target = fopen(target_file, "w");
   if( target == NULL )
   {
      fclose(source);
      printf("Press any key to exit...\n");
      exit(EXIT_FAILURE);
   }
   while( ( ch = fgetc(source) ) != EOF )
      fputc(ch, target);
   printf("File copied successfully.\n");
   fclose(source);
   fclose(target);
   return 0;
}
```

Example 3: Programme to delete a file.

This 'C' programme deletes a file, which is entered by the user; the file to be deleted should be present in the directory in which the executable file of this programme is present. Extension of the file should also be entered. Note that deleted file does not go to recycle bin, `remove` macro is used to delete the file. If there is an error in deleting the file then the error will be displayed using `perror` function.

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

```
    char file_name[25];
    printf("Enter the name of file you wish to delete\n");
    gets(file_name);
    status = remove(file_name);
    if( status == 0 )
       printf("%s file deleted successfully.\n",file_name);
    else
    {
       printf("Unable to delete the file\n");
       perror("Error");
    }
    return 0;
}
```

Example 4: Programme that writes results into an output file.

```
#include <stdio.h>
void main()
{
FILE *fp = fopen("D:\\f_read1.txt","w");
int i;
for(i = 0;i < 5; i++)
   fprintf(fp,"i=%d\n", i);
fclose(fp);
}
```

Output

```
i=0
i=1
i=2
i=3
i=4
```

Eample 5: Programme to read an array of ten integers from the console and to write the entire array into a file.

```
#include<stdio.h>
int main()
{
   FILE *p;
   int i,a[10];
   if((p=fopen("E:\\myfile.txt","wb"))==NULL)
   {
```

```
        printf("\nUnable to open file myfile.dat");
        exit(1);
    }
    printf("\nEnter ten values, one value on each line\n");
    for(i=0;i<10;i++)
        scanf("%d",&a[i]);
    for(i=0;i<10;i++)
        fprintf(p," %d ", a[i]);

    fclose(p);
    return 0;
}
```

Output:

```
Enter ten values, one value on each line
11
21
34
54
23
78
37
81
27
61
```

Contents of output file: `myfile.txt`

```
11  21  34  54  23  78  37  81  27  61
```

****** ☺ ******

DISCLAIMER:

The information on this website does not warrant or assume any legal liability or responsibility for the accuracy, completeness or usefulness of the courseware contents.

The contents are provided free for noncommercial purpose such as teaching, training, research, extension and self learning.

****** ☺ ******