U.P.Rajarshi Tandon Open University, Allahabad

UGCS-102
Problem Solving Through "C"

Block 1

Introduction to Algorithms and Program Design

Printed and Published by Vinay Kumar, Registrar, Uttar Pradesh Rajarshi Tandon Open University, 2024.

SUSHI-017

Printed By: Chandrakala Universal Pvt. Ltd. 42/7 Jawahar Lal Neharu Road, Prayagraj.

Block-1 COURSE INTRODUCTION

The objective of this course is to introduce the basic problem solving techniques as well as introductory programming using 'C' language. By learning the problem solving methods, you should be able to analyze a problem and develop an algorithm for its solution. The aim is to provide an extensive variety of topics on this subject with appropriate examples. The course is organized into following blocks:

Block 1 introduces the notion of algorithms and fundamental principles of program design.

Block 2 covers the basic features of 'C' programming language.

Block 3 describes the Control features of 'C' programming language.

Block 4 describes the advanced features like pointers, arrays, structures, functions and Input/output operations on file.

UNIT-1 Introduction to Algorithms

Structure

1.0 INTRODUCTION

In this unit, the focus is to analyze the problem and develop an algorithm for its solution. The problem solving methods are discussed in detail to provide the insight view. Algorithms and flowcharts are two common modes of solving a problem and finally how the algorithm and flowcharts are converted into a 'C' code is illustrated in this unit

1.1 OBJECTIVES

After the end of this unit, you should be able to:

- analyze the problem and develop an algorithm for its solution;
- represent an algorithm in an abstract language (eg. pseudo-code, Structure Diagrams);
- represent an algorithm with the help of flowchart.
- understand the fundamental principle of program design.

1.2 PROBLEM SOLVING TECHNIQUES

The ability to solve a problem is developed with time. When you try to solve a smaller and simpler problem, you become more attentive about the steps required for writing solutions of the problems. The two approaches for problem solving are analytical approach and algorithmic approach.

Analytical approach: This approach is used mainly by mathematicians and physicists. In this approach, we try and solve a problem by:

- separating the given quantities
- identifying what is to be solved

- applying correct formulae for solution
- performing operations such as addition/subtraction (if necessary)
- finally getting an answer

Example 1: What is the volume of a cuboid if length, breadth and height are 3, 4 and 5 units respectively?

Here, length, breadth and height are 3, 4 and 5 units. Let volume of a cuboid be *vol*.

We know that volume of a cuboid is the multiplication of length, breadth and height. So,

vol = length * breadth *height

substituting the values of length, breadth and height, we get;

$vol = 3*4*5 = 60$ units^3

Algorithmic approach: This approach uses a sequence of deterministic steps to solve problems. An example is the instructions needed to prepare a cup of tea:

1. Boil half cup water
2. add half cup milk in the kettle
3 add one teaspoon sugar in the kettle
4. add half teaspoon tea leaves in the kettle
5. boil the solution for three minutes
6. empty the kettle in the cup

The order of the instructions is important. This is very similar to computer processing

- The *Inputs* (Steps 1 to 4) are the ingredients
- The instructions (Step 5)describe the *Processing*
- The *Output* (Step 6) is a tea

Check your progress 1

The recipe above is not exactly an algorithm. Why?

In nutshell, problem solving through computers is done by performing following steps:

- Clearly define the problem.
- Identify the input(s) for solving the problem.
- Examine or analyze the given problem and formulate a method to solve it. Split the problem into a sequence of elementary tasks (if required).
- Devise an *algorithm* to solve the problem.
- Make *flowchart* of an algorithm to know the control flow.
- Write an algorithm (or program) in any programming language.
- Compile and run the program (debugging).

- Test the program (debugging).
- Collect output.

The devise phase of an algorithm (see step four above) normally consists of

a) a sequence of instructions (**sequence**)

b) an ability to make choices (**selection**)

c) an ability to repeat parts of the process (**repetition**)

a) Sequence: It represent that each of the instructions should be numbered, and we normally execute the next instruction in the sequence. For e.g.

1. Take radius of a circle as input
2. Calculate area of circle.

Here, note that sequence of instruction is important. Without knowing the radius, we can't calculate the area of a circle

b) Selection: The selection has an ability to make choice among various existing choices. We may think of using an IF statement.

```
IF something is true
THEN do an action
ELSE do another action
ENDIF
```

```
For e.g. IF age >=18
             THEN Write("you are adult")
         ELSE
             Write("you are minor")
         ENDIF
```

c) Repetition: Sometimes, we need that part of our algorithm repeats till the condition is true. To do this, we may use WHILE, FOR or REPEAT UNTIL construct.

```
    WHILE something is true
        keep on doing an action
ENDWHILE
```

```
REPEAT
    keep on doing an action
UNTIL something is true
```

1.3 ALGORITHM

Definition: An algorithm is a well defined *finite sequence* of *executable instructions* with these properties:

- (Input): There may be zero or more external inputs supplied to an algorithm.
- (Output): At least one output quantity is generated by the algorithm.
- (Definiteness): There is no ambiguity in any instruction and each instruction should be clear.
- (Effectiveness): Each instruction must be sufficiently basic so that it can be easily written in any programming language.
- (Finiteness): In all cases, the execution of an algorithm must terminate atter a finite number of steps.

The efficacy of an algorithm is examined on the basis of following parameters:

1. **Time complexity:** It is the amount of computer time required by an algorithm for its execution. This time includes the time for compilation of a program as well as time for execution.

2. **Space complexity:** It is the amount of space (or memory) required by an algorithm for its execution and outcome of final output.

An algorithm may be expressed in terms of pseudo codes and flow charts, which we discussed in next unit.

1.4 Examples

Now, let us write algorithms for some example problems.

Problem: Write an algorithm for swapping (interchange with each other) of two numbers.

Solution: We can swap (or interchange) two numbers by using a temporary number Z. If we assign value of X to Z, value of Y to X and value of Z to Y, than both numbers are interchanged. Re-writing the above statement in the algorithm form as:

Algorithm 2 (Swapping of two numbers)

```
1. Read two numbers X and Y
2. Perform
   (a) Z=X
   (b) X=Y
   (c) Y=Z
3. Display X and Y
```

Check Your Progress 2:

Can you get better solution for above example?

Problem: Write an algorithm for finding the factorial of a given positive number.

Solution: We know that factorial of a given number is calculated as:

N! = 1.2.3.4............ (N-2). (N-1)

and base condition is 0! =1

```
Algorithm 3 (Calculating Factorial of a given number)
1.    Read positive number N
2.    Set Factorial = 1
3.    WHILE N ≥D
          (a) Factorial = Factorial * N
          (b) N = N -1
4.    ENDWHILE
5.    Display Factorial
```

Problem: Write an algorithm for finding the greatest common divisor (GCD) of two numbers.

Solution:

```
Algorithm 4 (Finding GCD of two numbers)
1.    Read A and B as larger and smaller of two input numbers
2.    Divide A by B and call the remainder R
3.    IF R is not equal to 0 THEN
          (a) A=B
          (b) B =R
          (c) return to step 2
4.    ELSE
          GCD = B
5.    ENDIF
```

Check Your Progress 3:

What will when if A and B are equal?

Problem: Write an algorithm for a program which will sum up and display the following:

Sum = 1 + 4 + 7 + 10 + 13

Solution: By analyzing the above series, we note that first term is 1 and every next term is obtained by adding 3 to the previous term. This addition continues till we get last term 13. Using this approach, we write an algorithm 2 for the above problem.

```
Algorithm 1 (uses GOTO statement)
1.    set sum to 0
2.    set X to 1
3.    add X to sum
```

```
4.    Increase X by 3
5.    If X is less than or equal to 13 GOTO line 3
6.    display sum
```

Note that Steps 3 and 4 are repeating statement (Looping statement), and

Step 5 is a selection operation (Selection statement).

Another way to solve the same problem is carried out with the help of WHILE.... ENDWHILE structure (see Algorithm 2).

Algorithm 2 (uses WHILE .... ENDWHILE structure)

```
1.    set sum to 0
2.    set X to 1
3.    WHILE X is less than or equal to 13
4.        add X to sum
5.        Increase X by 3
6.    ENDWHILE
```

At last, another way to solve the same problem is carried out with the help of FOR..... ENDFOR structure.

Algorithm 3 (uses FOR .... ENDFOR structure)

```
1.    set sum to 0
2.    FOR X going from 1 to 13 with steps of 3
3.        add X to sum
4.    ENDFOR
5.    display sum
```

Note: Algorithms are not necessarily unique. Several completely different algorithms can solve the same problem.

Problem: Design an algorithm for finding the sum of the digit of a number.

Solution:

Algorithm 4 (uses WHILE .... ENDWHILE structure)

```
1.    Read number N
2.    Set sum = 0
3.    WHILE N   0
      (a)    Remainder = N % 10
      (b)    Sum = Sum +Remainder
      (c)    N = N/10
      ENDWHILE
4.    Display Sum
```

Check your progress4

1. *Design an algorithm for finding the reverse of a number.*
2. *Describe an algorithm which will accept two numbers from the keyboard and calculate the sum and product displaying the answer on the monitor screen.*
3. *Design an algorithm for generating a Fibonacci series up to n terms.*

   Write an algorithm to check whether the given number is a prime number or not.

1.5 Summary

In this unit we presented problem solving techniques to analyze a problem and showed how to formulate a good solution using algorithms. We have also described the essential properties that an algorithm should have. The two parameters, time and space complexity, for analyzing an algorithm was also discussed with enough examples so that readers should be well acquainted. At last, section 1.4 covers some example of algorithms to solve simple problems.

UNIT 2: Pseudo-codes and Flowcharts

Structure

2.0 INTRODUCTION

In previous unit of this block, you learned about algorithms and their properties. We have also seen some examples illustrating how to build an algorithm for the given problem. An algorithm cannot run on a computer, you have to write the algorithm in a programming language that a computer can understand. The languages that a computer understands are known as computer programming languages. In this unit, we focus on two famous tools for representing an algorithm, namely flowchart and pseudo code. We will try to learn and use both Flowchart and Pseudo-code techniques while we are designing algorithms (finding solutions) for the given problems. Flowchart and Pseudo code actually give better understandability for the problem solution.

2.1 OBJECTIVES

This unit deals with the representation of algorithm in terms of flowchart and pseudo code.

At the end of this unit, you will be able to:

- Explain the need of flow charts.
- Draw flow chart for the given problem.
- Write pseudo code using various constructs.
- Understand the three basic logical structures, namely, sequence, selection and repetition.

2.2 Tools of Algorithm

There are various Algorithm tools available today to properly understand the problem solving process, but the most common tools are Flowcharts and Pseudo-Codes. In this unit, we provide the notion of Flowcharts and Pseudo-Codes as an alternative way for writing algorithms (finding solutions) for the given problems.

2.3 Pseudo codes

Pseudo codes or *Structured English* are also known as *Structured Pseudo codes*. There are no general accepted standards for pseudo-codes. We will work with a form that has minimum number of rules and is essentially language independent. Since pseudo-code instructions are written in English, they can be easily understood and reviewed by users. Because structured pseudo codes come into view to be fairly factual translations of algorithm, they closely resemble the refined product.

Pseudo codes have the following properties:

(a) It is similar to spoken English rather than normal programming language-PASCAL, BASIC. So it is easily understand by programmers and non-programmers.

(b) Its vocabulary set is much restricted than normal speech, as it has to follow a rigid logical order.

(c) There are number of conventions for writing Pseudo codes.

Structured pseudo code uses keywords (e.g. IF… ELSE, WHILE, FOR, COMPUTE, MAX, MIN *etc.*) which by some conventions, are written in capital letters and have a specific logical meaning in the context of the description. We have already seen above keywords while designing algorithms in Unit 1.

The key to good algorithm design and thus to programming lies in limiting the control structure to only three constructs. Since pseudo codes are a way to represent an algorithm, they have three basic logical structures.

a) SEQUENCE

b) SELECTION

c) REPETITION

a) SEQUENCE

It correspond that each of the instructions in the sequence should be numbered, and each instruction should be executed in a sequence. You can write anything in small case letters for assignments, data manipulations, initialization etc.

Example 1:

```
1.  Input the marks for SUBJECT1, SUBJECT2 and SUBJECT3
2.  Total_marks = SUBJECT1+SUBJECT2+ SUBJECT3
```

Here, order of instructions are important because without knowing the marks of three subjects, we can't calculate the total marks.

Check your progress

1. Write the pseudo code for finding the average of six numbers, and sum of the numbers is given

a) SELECTION

Most of the programs need a number of '*choices*' where the subsequent action depends on the choices being made in structured English.

There are four types of SELECTION structures:

1. IF ...ENDIF (Simple IF) Structure
2. IF....THEN....ELSE ENDIF Structure
3. CASE ... OF ....ENDCASE Structure
4. NESTED IF ..... ELSE Structure

Most common construct for selection is IF....THEN.... ELSE statement.

```
IF something is true
        THEN do an action
ELSE do another action
        ENDIF
```

For example, a company offers 20% discount to their premium customers, 10% discount to their standard customers and 5% discount to their regular customers. The Pseudo code representation of this statement is as follows:

```
IF Customer is a premium Customer
        THEN give 20% discount
ELSE IF Customer is a standard Customer
        THEN give 10% discount
ELSE IF Customer is a regular Customer
        THEN give 5% discount
ELSE
        no discount is given
ENDIF
```

Now suppose we have to make a more difficult decision. At this time, company provides 30% discount to their premium customers who have been customers for last two years, 20% discount to their standard customers who have been customers for last one year, but other customers get only 10% discount.

```
IF Customer is a premium Customer
        IF Customer is over 2 years
                THEN 30% discount is given
        ELSE 10% discount is given
ELSE IF Customer is a standard Customer
        IF Customer is over 1 years
                THEN 20% discount is given
        ELSE 10% discount is given
    ELSE
        10% discount is given
    ENDIF
```

Another type of decision is a CASE statement which is a substitute to the IF..... THEN..... ELSE construct illustrated above. The CASE statement provides relatively simple decisions and can be used in place of IF...THEN....ELSE to avoid any confusion among the programmers. For example, we can write the above difficult decision with the help of CASE statement.

```
CASE 1: Customer is a premium Customer OF
    IF Customer is over 2 years
        THEN Give 30% discount
    ELSE 10% discount is given
CASE 2: Customer is a standard Customer OF
    IF Customer is over 1 years
        THEN Give 20% discount
    ELSE 10% discount is given
default: 10% discount is given
ENDCASE
```

b) REPETITION

Sometimes we require that a block or set of instructions may be repeated until a final condition is reached or the given condition is not satisfied. For example assume that Block 1 is the name of the block consisting of several instructions. We desire that this set of instructions (Block 1) is to be executed until the number of processed records reached 10. In structured pseudo code, the four construct for repetitions are:

1.	**WHILE** condition is true Block 1 **ENDWHILE**	This type of conditional loop tests for terminating condition at the *beginning* of the loop. In this case no action is performed if the first test causes the terminating condition to evaluate as false.
2.	**DO** Block 1 **WHILE** condition is true	This type of conditional loop This type of conditional loop tests for terminating condition at the *end* of the loop. This loop executes at least once even if the terminating condition is initially false, so iteration time is greater than and equal to 1.

3.	**REPEAT** Block 1 **UNTIL** condition is true	This loop executes at least once even if the condition is initially TRUE, so iteration time is greater than and equal to 1.Iterations are carried out while condition is *FALSE*, and *stopped* when the condition is *TRUE*.
4.	**FOR** (starting state, stopping condition, increment) Block 1 **ENDFOR**	FOR loop is used when the *number of iterations is known in advance*. This, in its simplest form, uses an *initialization of the variable as a starting point*, a *stop condition depending on the value of the variable*. The *variable is incremented on each iteration* until it reaches to the required value.

Example 2: Write a program segment that repeatedly asks for entry of a number in the range 1 to 100 until a valid number is entered.

Solution:

```
REPEAT
        PRINT "Enter a number between 1 and 100"
        ACCEPT number
UNTIL number < 1 OR number > 100
```

Example 3: Write pseudo code to print out each character typed at a keyboard until the character 'x' is entered.

Solution:

```
WHILE letter <> 'x'
        ACCEPT letter
        PRINT "The character you typed is", letter
ENDWHILE
```

Note: *The third construct REPEAT.... UNTIL is not available in C language.*

Example 4: Write a pseudo code to print 1, 2, 3, 4.

Solution:

```
FOR (n = 1; n <= 4; n=n + 1)
        PRINT n
ENDFOR
```

In section 2.5, we will see how to use various selection and repetition statements for solving a given problem.

2.4 Flow charts

Like Pseudo-codes, flowchart is another way to represent an algorithm. A flowchart is consisting with set of standard symbols, each of which is distinctive in shape and represents a particular type of operation (see Table 1). The symbols are joined by straight lines called *flowlines*. These flowlines are actually arrows to specify the order in which the operations are performed.

A flowchart is useful in many situations where complex programs contain numerous branches, since it can describe the interrelationships between the various branches and loops. Flowcharts also permit us to rapidly test several alternative solutions to a problem since it is much easier to draw the flowchart than to write the program. Once a flowchart has been draw, the task of writing the program becomes simplify.

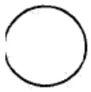
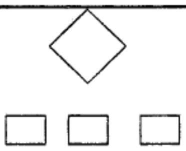
Finally, a flowchart is an admirable medium for documenting a program. It provides a suitable means of communication between both programmers and nonprogrammers. This is important during the design of a program, especially when numerous people are working on the same task. Since a flowchart is not reliant on a particular programming language, it can be understood by another programmer and by people who have partial knowledge of programming. This can be of immense advantage during later maintenance and use of the program.

Flowchart Symbols

The symbols used in flowcharts are standardized by the American National Standards Institute. If the direction of flow is not clear, then arrows are used on the connecting flowlines. A flowchart should have one beginning (or start) and one or more end (or stop) points and should be arranged so that the direction of computation is from top to bottom and from left to right. It may be possible that flowlines can cross each other, but the crossing flowlines should be independent of each other. Whenever possible, crossing of flowlines should be avoided, since it makes the flowchart difficult to read. Table 1 shows the various flowchart symbols and the operation that they represent. While there are so many shapes tor specific purposes, to avoid complexity, in this course, only a limited subset of these shapes will be shown and going to be used in examples. Next, we describe the meaning of various symbols.

Tabel 1: Flowchart symbols

Symbol	Name	connotation
	Flowlines	Represent direction of processing

	Terminal	An oval is used to indicate the beginning and end ot a program
	Input/Output	A parallelogram indicates the input or output of information.
	Process	Represent computations or data manipulations/ processing
	Decision	Represents a decision point in the process, usually requiring a 'yes' or 'no' response, then branching to different parts of the flowchart.
	On-page connector	Connects two or more processes into one. Represents connection with another process. A reference to the new process should appear within the circle.
	Off-page connector	connector Connects flows on different pages.
	Program Decision- 3 Or More Options	Represents multiple choices for the user, with the selected option determining the user's path through the rest of the program.
	Loop	A hexagon indicates the beginning of a repetition structure.

Flow charts also have three basic logical structures:

a) SEQUENCE

b) SELECTION

c) REPETITION

a) SEQUENCE

Initialization, data manipulation (or computation) and assignment statements are considered as Sequence Statements and they are represented with PROCESS symbol (Rectangle) in FLOWCHARTs. Following are the examples of initialization and data manipulation statements.

```
Example :    INITIALIZATION
             pi = 3.14
             interest_rate = 10%
             amount = 5000
             year = 5
```

In above example, the values of pi, interest_rate, amount and year are initialized. The above sequence of statements are executed one by one.

b) SELECTION

There are four types of SELECTION structures:

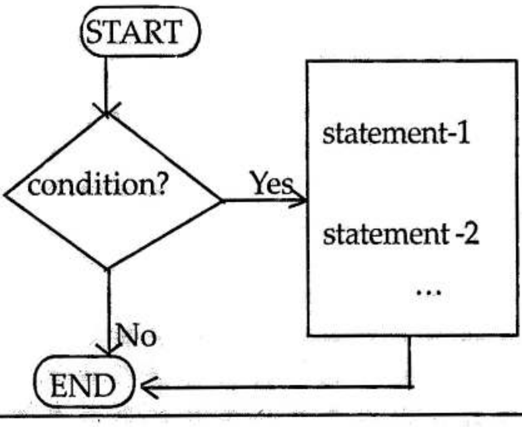
1. IF ...ENDIF (Simple IF) Structure
2. IF....ELSE......ENDIF Structure
3. CASE ... OF ....ENDCASE Structure
4. NESTED IF ..... Structure

Next, we show how to represent these structures using pseudo-code and flow charts.

1. IF ...ENDIF (Simple IF) Structure

Pseudo code **Flow chart**

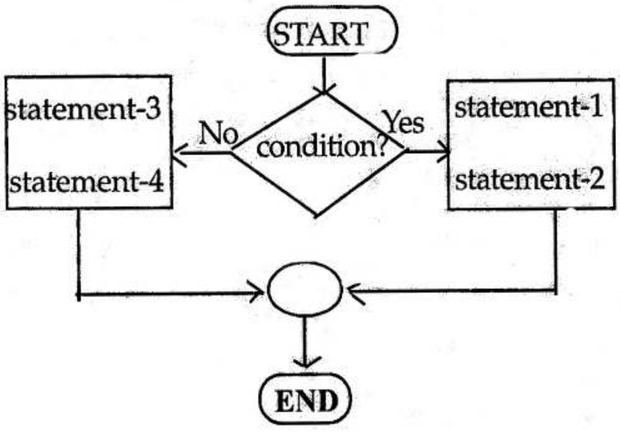
```
IF condition
      statement-1
      statement -2
      ....
      ....
END IF
```



2. IF....ELSE......ENDIF Structure

Pseudo code **Flow chart**

```
IF condition
    statement-1
    statement -2
    ...
    ...
ELSE
    statement-3
    statement -4
    ...
    ...
END IF
```

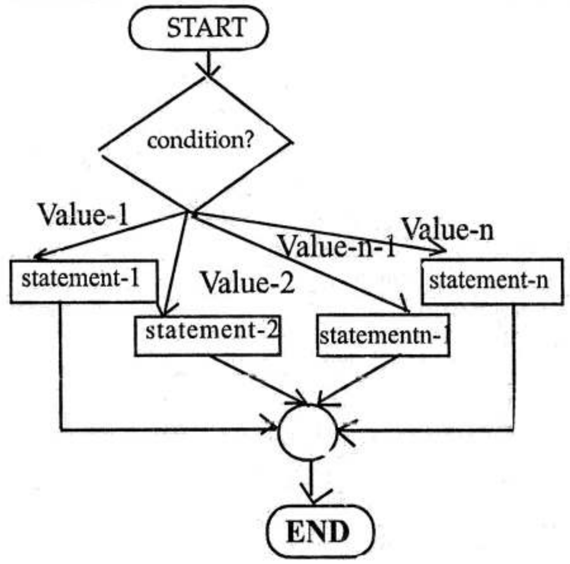


3. CASE....OF......ENDCASE Structure

Pseudo code

```
CASE field OF
 Value-1: statement-1
 Value-2: statement -2
 ...Value-n: statement -n
 END CASE
```

Flow chart



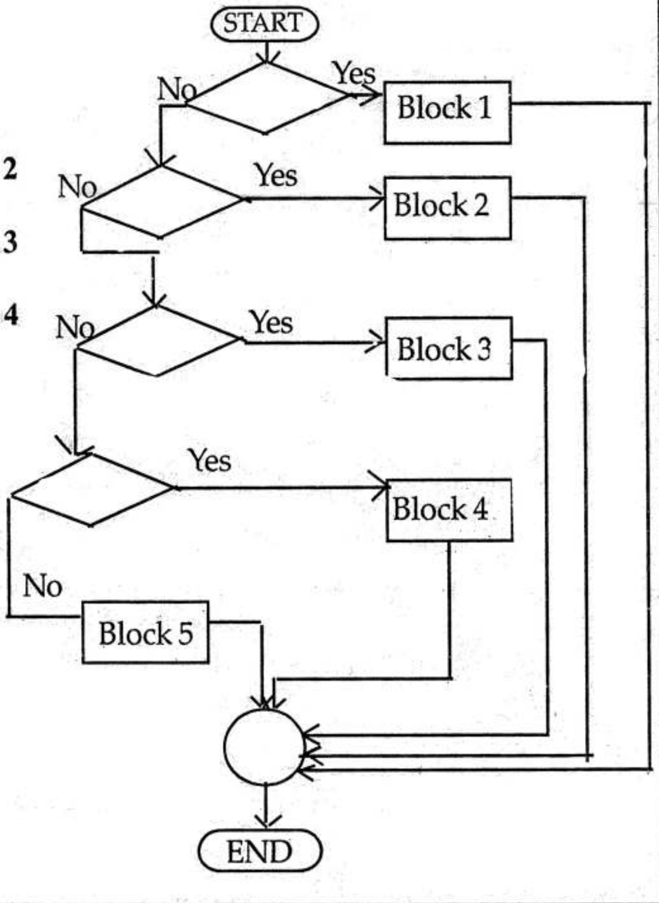
Note: *CASE statement can be used in place of Nested IF statements when choices are specific values and they are not including ranges.*

4. NESTED IF Structure

Pseudo code

```
IF condition1
 Block 1
ELSE IF condition 2
 Block 2
ELSE IF condition 3
 Block 3
ELSE IF condition 4
 Block 4
ELSE Block 5
ENDIF
ENDIF
ENDIF
ENDIF
```

Flow chart



Check your progress

1. Write the pseudo code using nested if statement for student grades with following conditions:

    if grade is greater than or equal to 85 then print "Grade A"
    if grade is greater than or equal to 70 then print "Grade B"
    if grade is greater than or equal to 60 then print "Grade C"
    if grade is greater than or equal to 50 then print "Grade D"
    if grade is below 50 then print "Fail"

2. Convert the above pseudo code using CASE statement.
3. Draw the flow chart for question number 1 and 2.

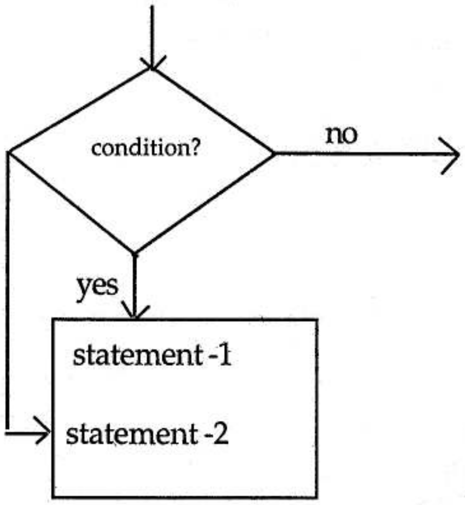
a) REPETITION (Control Structure)

A repetition structure represents some part of the program that repeats. This type of structure is also known as loop or control structure. There are four different LOOP Structures:

1. WHILE ...... ENDWHILE loop
2. DO .... WHILE loop
3. REPEAT......UNTIL loop
4. FOR ...... ENDFOR loop

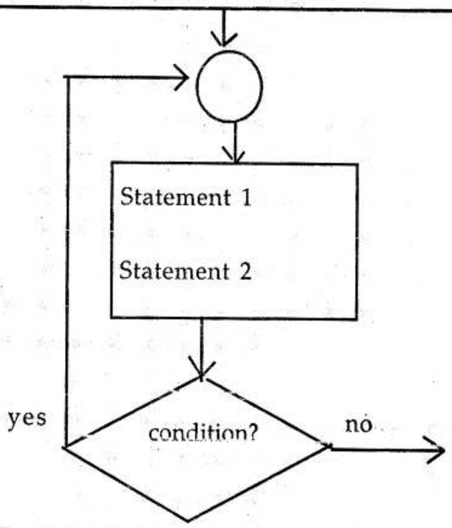
Next, we present representation of these loop structures using pseudo-code and flow charts.

1. WHILE ...... ENDWHILE loop	
Pseudo code	**Flow charts**

WHILE condition
    Statement -1
    Statement-2
    ...
ENDWHILE
	

2. **DO...... WHILE loop**

Pseudo code	**Flow charts**

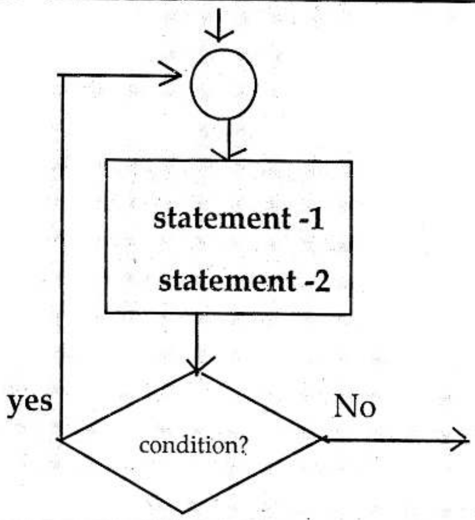
Do
    Statement -1
    Statement-2
    ...
WHILE condition
	

3. **REPEAT...... UNTIL loop**

Properties:

- This loop executes at least once even if the condition is initially TRUE, so iteration time is greater than and equal to 1.
- Iterations are carried out until the condition remains, and stopped when the condition is FALSE.
- Working method is: First Execute and Then Check the condition

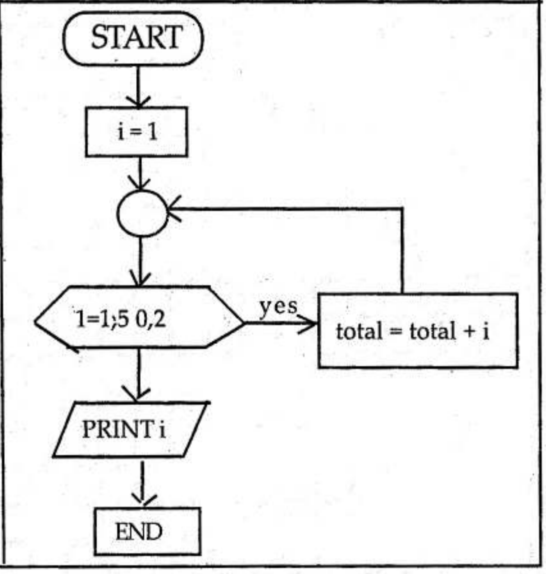
Pseudo code	**Flow charts**

REPEAT
    Statement -1
    Statement-2
    ...
UNTIL condition
	

4. FOR (starting state, stopping condition, increment).....**ENDFOR loop**

Pseudo code	**Flow charts**
Example	

```
set i to 1
FOR i = 1 to 50 with step 2
        add i to total
END FOR
PRINT total
```



So far we have seen how to draw flow chart and write pseudo codes. Next, we present some problems so that you can become more familiar with these tools of algorithm.

2.5 EXAMPLES

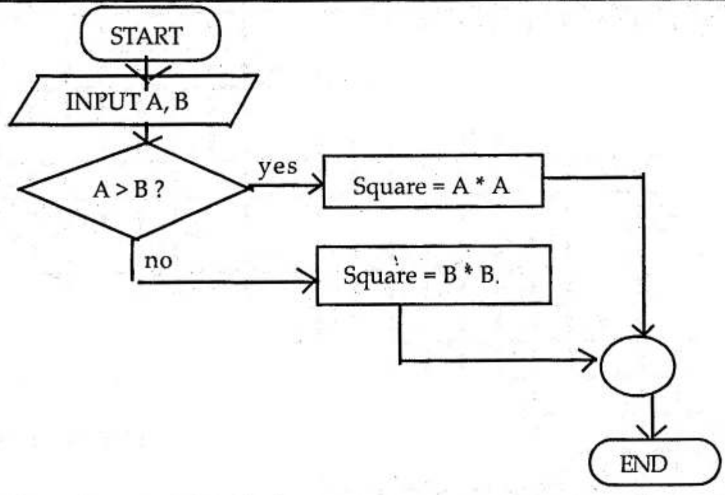
Problem: Write the pseudo code and draw flow chart to accept two numbers and find the square of the biggest number.

Solution: The pseudo code and flow chart for the above problem is given below:

Pseudo code

```
INPUT A,B
IF A is greater than B
        THEN square = A * A
ELSE square = B * B
ENDIF
```

Flow chart



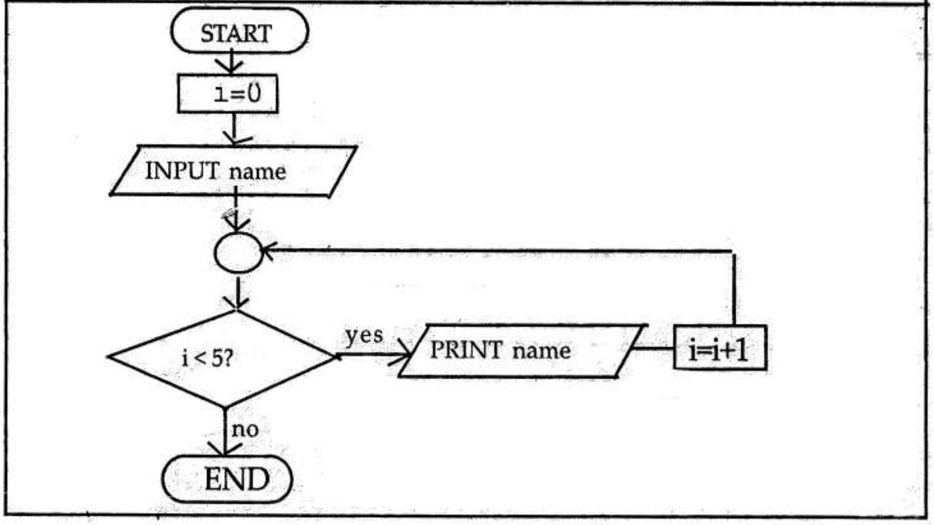
Problem: Write the pseudo code and draw flow chart to print your name 5 times.

Solution:

Pseudo code

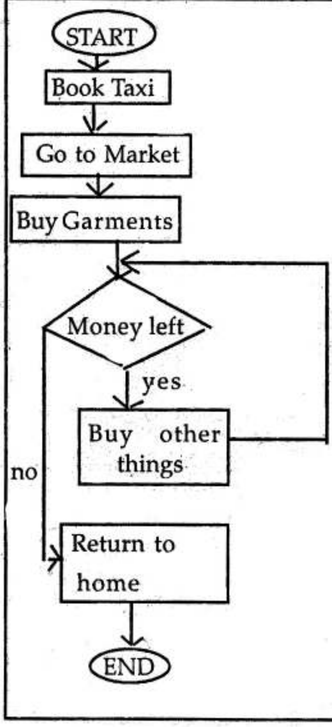
```
 Set i=0
 INPUT Name
WHILE i is less than 5
      PRINT name
      increment i by 1
END WHILE
```

Flow chart



Check your progress

1. Write the pseudo code and draw flow chart to choose the largest number from a set of three numbers, A, B, C.
2. Write the Pseudo code for the following chart.



Check your progress

1. Draw the flowchart for the following pseudo code.

```
START
    INPUT Num1
    INPUT Num2
    PRINT menu of operations
    INPUT operation
    WHILE (operation < 1) and (operation > 4)
        PRINT error message
        INPUT operation
    ENDWHILE
    CASE operation OF
    1 : Result = Num1 + Num2
    2 : Result = Num1 – Num2
    3 : Result = Num1 * Num2
    4 : IF Num2 is not equal to 0
            THEN Result = Num1 / Num2
            ELSE
                Result = 0
            ENDIF
    ENDCASE
    PRINT Result
END
```

2. The sequence of Fibonacci numbers is defined as below:

$$f(i) = f(i-1) + f(i-2) \text{ with } f(0) = 1 \text{ and } f(1) = 1$$

Write the pseudo code and draw a flowchart to calculate and display Fibonacci numbers.

3. (a) List the main keywords used in Pseudocodes. (b) What control structures they represent.

4. Design an algorithm and the corresponding flowchart for finding the sum of the numbers 2, 4, 6, 8, …, n

5. Using flowcharts, write an algorithm to read 100 numbers and then display the sum.
6. Write an algorithm to read two numbers then display the largest.
7. Write an algorithm to read two numbers then display the smallest
8. Write an algorithm to read three numbers then display the largest.
9. Write an algorithm to read 100 numbers then display the largest.

2.6 SUMMARY

In this unit, we studies about pseudo code and flow charts. We have also seen that both are the alternative ways of defining the problem solving process in sequential manner. Both of these methods uses three logical structures, sequence, selection (or decision) and repetition (control structure) for solving any problem. You also learned that:

- what is program flow chart?
- what symbols and constructs are used in drawing flow charts
- what are the guidelines for drawing flow charts
- how to draw a good flow chart
- how to use nested loops in flow charts
- how to use multiway selection in flow charts
- how to use nested loops in flow charts
- how to use multiway selection in flow charts

UNIT 3: PROGRAM DESIGN PRINCIPLES

Structure

3.0 INTRODUCTION

In this unit, we will see the basic design principles of a good program and also describe some terminology related to the computer programming. The unit includes the introduction of programming languages and showing that how the problem is solved by going through a programming cycle. The programming paradigms like, unstructured programming, structured programming, procedural programming, modular programming, top down and bottom up design are also discussed.

3.1 OBJECTIVES

Program designing is a very crucial step in software development process; hence it is not so easy to develop good software without the proper understanding of programming paradigms. At the end of this unit, you will be able to:

- know about the various programming languages, including low level and high level language.
- Understand the definition of program and the principles of good programming.
- Know the concept of programming cycle and its various phases to develop a good program.
- Know different programming approaches.

3.2 INTRODUCTION TO COMPUTER PROGRAMMING

A computer is an information processor. Data and information are given as **input** into the computer and then processing is performed to produce **output** (see Figure 3.1).

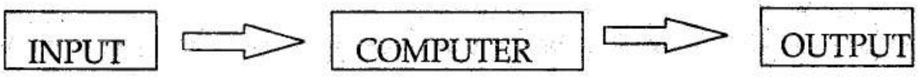


Figure - 3.1 Structure of Computer

All the physical components including input and output devices that are attached to a computer are known as **hardware.** The set of instruction performed by the computer is called a **computer program.**

Computer is a sense less machine and it cannot perform any work without instruction from the end user. The tremendous speed and accuracy for performing the instruction is the key property of computer. It is up to you to make a decision that what you want to do and in what sequence. That's why computer cannot take its own decision as you can. Therefore, it needs specific *logically related instructions* that the programmer feeds into a computer to solve a particular problem. These instructions are termed as **program.** The set of programs written for a computer is referred to as **software**. A **programming language** is a set of convention that provides a technique to tell the computer what operations it has to perform.

A programming language used to communicate with the computer has certain specific characteristics. It has a restricted set of vocabulary. Each "word" in the programming language has clear-cut meaning. Every programming language has certain limitations but still they are used in gradual way to solve difficult problems.

There are two major types of programming languages; *low – level* and *high-level* language.

a) Low-level Languages

Both machine and assembly level languages are called "low-level". The term "low" does not mean "inferior" in any sense, both rather "closeness" to the way in which the machine has been built.

Machine-Level Language

Although computers can be programmed to understand many different computer languages, there is only one language which is understood by the computer without using translation or interpretation.

An instruction prepared in any machine language has a two – part format. The first part is the command or operation, and it tells the computer what function is to perform. Every computer has an operation code or opcode for each of its functions. The second part of the instruction is the operand, and it tells the computer where to find or store the data or other instruction that are to be manipulated.

Programs written in machine language can be executed very fast by the computer. This is mainly because machine instructions are directly understood by CPU and no translation of the program is required. Because the internal design of every type of computer is different from every other type of computers and needs different electrical signals to operate, the machine language also is different from computer to computer. Therefore the programs or instructions written in machine language are machine dependent. Hence the program which is running on a machine can't run on the other machine.

Assembly –Level Language

One of the first steps in improve the program preparation process was to substitute letter symbols mnemonics for the numeric operation codes of machine language. The language which substitutes letters and symbols for the numbers in the machine level language program is called *an assembly level language* or *symbolic language.* A program written in symbolic langrage that uses symbols instead of numbers is called an *assembly code.*

This assembly program also enables the computer to convert the programmer's assembly language instructions in to its own machine code. A program of instructions written by a programmer in an assembly language is called the *source program*. A source program has been converted into machine code by the *assembler* (see Figure 3.2). This machine code is referred as object program.

Assembly-level language has number of advantages over machine-level language, some of them are: it is easier to understand and use, easy to locate and correct errors, easier to modify.

But there are certain limitations of assembly-level languages like it includes machine dependency (designed for the specific make and model of processor), knowledge of hardware to the programmer for writing the code.

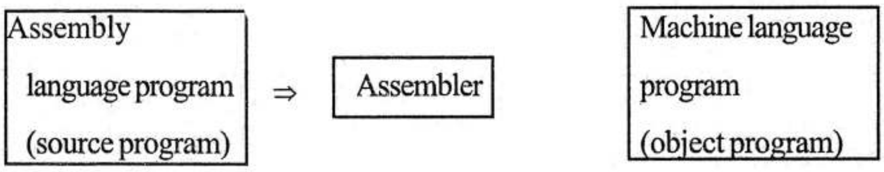

Assembly language program (source program)
⇒
Assembler
Machine language program (object program)

Figure: 3.2 - Assembler

b) High level language

Writing a program in machine language or assembly language requires a deep knowledge of the internal structure of the computer. While writing a program in any of these language, a programmer has to remember all the operation codes (numeric or mnemonics) of the computer and know in detail what each code does and how it affects the various registers of the computer. Therefore, to make programming task convenient for the programmer and easier to understand for the user the high level languages were developed..

High-level language are intended to be machine independent and are problem oriented languages (POLSs) i.e. they reflect the type of problem solved rather than the features of the machine. These languages enable the programmers easier for him to concentrate on the logic to solve the problem. Advantages of high-level language are easier to learn, less time to write, machine independent simplification and diagnostic error detection.

There are many programming language that are used to create programs. Some of the examples of these languages are **BASIC, FORTRAN, COBOL, Pascal, C, C++**. Like English language, all programming languages have a rule of grammar of their own known as **syntax** of the language.

Since computer hardware is capable of understanding only machine level instructions, so that a software is required to convert the instruction of a program written in high-level language (source code) to machine instructions (object code) before the execution of program. We have seen that assembler also perform this conversion process but for the high level programming languages, interpreter or compiler is required for this purpose. It is shown in Figure 3.3.

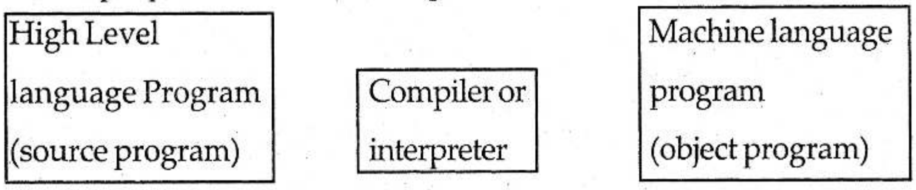

High Level language Program (source program)
Compiler or interpreter
Machine language program (object program)

Fig. 3.3 : Compiler or interpreter

Compiler

Compilers are large programs which reside permanently a secondary storage. When the translation of a program is to be done, they are copied into the main memory of the computer. The compiler, being a program, is executed by the CPU. While translating a given program, the compiler analyses each statement in the source program and generates a sequence of machine instructions which when executed, will precisely carry out the computation specified in the statement. As the compiler analyses each statement it uncovers certain types of errors. These are referred to as diagnostic errors. A compiler cannot diagnose logical error. It only diagnoses grammatical (**syntax**) errors in the program.

Interpreter

An interpreter is another type of translator used for translating high-level language into machine code. It takes one statement and translates it into an machine instruction which is immediately executed. The translation and execution alternate for each statement in the high-level language program.

This differs from a compiler which nearly translates the entire source program into an object program and is not involved into its execution however in case of an interpreter, no object code is saved for future use because the translation and the execution processes alternate.

The performance comparison of both interpreter and compiler can seen as:

1. Interpreter provides fast response to change the source code.
2. Compiler is a complex program with respect to interpreter
3. Interpreters are easy to write and do not occupy much space in memory as compared to compiler.
4. The interpreter is a time consuming method because each statement must translate every timer it is executed from the source program.
5. A compiled machine language program runs much faster than an interpreter program.

Assemblers, compilers and interpreters are ***system software's*** *that* translate a source program written by the user to an object program which is meaningful to the hardware of the computer. These translators are also referred as ***language processor*** since they are used for processing a particular language.

3.3 PROGRAM DESIGN PRINCIPLES

Program designing is a very sedative phase of *software development cycle*. The prettiness of mind, skill of brain and sensible view is assorted with system objective to implement design.

The *designing process* is very complex (not simple), unwieldy and annoying with many curves in the way of doing well design.

The objectives of Program design are:

(i) **Substitute old system**: The new system is designed in such a way that it can be used to substitute old system because of high maintenance cost of old system and also the efficacy of old system is very low.

(ii) **Requirement of Industry**: Industry and organizations needs the development and installations of new system for the working groups and end user.

(iii) **Efficiency**: The new designed system is installed to raise the production of company or organization.

(iv) **Competition**: Every organization wishes to launch a new system that proves its strength and it is a matter of status also. In the era of thriving competition, if organization does not deal with modem technology, it is futile to face competitions.

(v) **Maintenance**: The new system is required to uphold organization position.

Individuals and computer industry repeatedly searches for more proficient ways to perform the software development process. One possible way to reduce the development cost and time is to standardize software programs and the programming process. The benefits of standardized programs are that they are easier to code, maintain, debug, and modify.

In latest years, a range of techniques have appeared attempting to reduce differences in the way programmers' design and develop software. A few of the most frequently used techniques for standardization are described as follows:

3.3.1 Program Specifications

Before writing a program, a detail specification must be prepared to show what exactly a program has to do. This task may be done by other programmer or by more senior programmer or a system analyst. It is the wastage of time if the programmer start typing the program before a clear specification has been created. Usually, such as a specification has three main parts:

a) **Input**: a detailed description of the format of the input data.

b) **Output**: a detailed description of the format of output data.

c) **Processing**: a detailed description for the processing of the program to get the output from the input.

3.3.2 Programming Circle

Before developing a solution for the given problem, it is not only sufficient to know the rules of a computer language beside this the problem-solving skills and techniques are also important. There are following five steps for program development to solve the given problem:.

a) Problem statement

b) Planning the solution or logic design

c) Program Coding

d) Program Testing

e) Documentation

a) Problem statement

It requires to define the problem precisely, and clearly as per the system requirements, i.e., kind of input, processing, and output required. Some other tasks those are also involved in this step can represent as:

- Understandability of the vocabulary used in the raw formulation?
- What and which type of information has been given?
- How can I identify a solution?
- What should I want to process and produce as output?
- Which information is missing (or left) and will any of this be of use?
- Is there any insignificant information?
- What assumptions should I made while writing a program?

b) Planning the solution or Logic design

To work out a given programming problem, one has to follow a systematic approach, i.e., devise an ordered set of activities that will convert a given input into the desired algorithm. Apart from these we should also concentrate on following two issues given below:

- Which mathematical structures seem best-suited for the problem?
- Are there any other problems that have been solved which resemble this one?

An algorithm consists of three main components. These are input, process and output. Algorithms implemented by a computer are known as **computer programs**. We have also discussed that a program consists of basically the following operations: -

I. Sequence — in order

II. Selection – select a choice based on some criterion (e.g., if…else)

III. Repetition — Iteration

These three operations are sufficient to describe any algorithm and we have also discussed that an algorithm can be described by drawing **flowcharts** and writing **pseudocode**.

c) Program Coding

We require a programming language to express an algorithm. Program coding refers to the process of transforming a *pseudocode* or *flowchart* into a computer program using a programming language. The program is written in accordance with the language syntax. There are many high-level programming languages, each with a different compiler, such as FORTRAN, COBOL, C, C++, Pascal, ADA, ALGOL, BASIC, etc.

There is a required set of programming rules to encourage the habit of following convention.

I. NAMES

Selection of variable names is very important during development of a program. It enables yourself as well as other programmers to debug and modify in near future. As the part of program documentation, names must clearly indicate both the kind of thing being named and its role in the program. There may be two choices, and you should prefer one and follow it consistently. Variable names may be one word long can begin either with a lower case or an upper case and thereafter are lower case including other acceptable characters.

Following are the guidelines for choosing meaningful names:

- Generally select English words, which explain the thing being named.
- Always use simple names for variables. For example, a loop counter might be a single letter.
- Avoid using lots of one- and two- character names like p1 or y.
- Use common prefixes/suffixes to correlate names of the same general type or category.
- Do not use misspell and meaningless suffixes to create variables with similar names, e.g. if the name *student* is used, do not define another variable with a name like *stu*, *stud*, or *stu1*.
- Do not use names that are analogous to each other and thus causes confusion.
- Avoid the use of acronyms as abbreviations, as a substitute, if your names are getting long you may eliminate vowels (e.g., lstElmnt), or use an unambiguous prefix (e.g., lastElem), but NOT le.

Here are some examples of good and bad variable (See Table 3.1)

Table 3.1: Examples of Good and Bad variable names

Good Name	**Bad Name**
TaxRate	Rate, tRate
Tax_rate	tr
Price_In_Dollar	pDollar, priceD
Max	Mx

I. Indentation

It is advisable that you should maintain consistency of indentation throughout your code. The proper indentation enables you and other programmers to read and understand the code in better way. Example of good indentation is given below.

```
IF condition
    THEN statements
ELSE
    statements
END IF
FOR (...)
    statements
ENDFOR
```

You can also use rightward drift for indentation. If you have deeply nested block of code, then indentation pushes you farther and farther to the right, so that no room is left to write a line of code. To keep away from this situation, you can use a modest size indentation. Four spaces are probably optimal.

II. Comments

Comments are very useful for the reader to debug, modify and understand your code. You can follow the given guidelines to add a comment:

- All internal documentation must be written in English. Do not use comments that simply put another way your code in English, without abbreviating it or adding any information.
- For each variable, provide a one-line comment about the use of the variable. These may follow the variable declaration as

```
Set i=0        /* loop variable */
FOR i = 1, i <10, step 1
/* Do some operation here */
END FOR
```

- you may also use a comment block to identify the use of several variables together.
- Give the purpose of major section of the program.

III. Careful use of White Space

Sometimes the simple insertion of a blank line makes the program much easier to read. But don't put too much white space otherwise the reader will not be able to see enough of the code at a time.

a) Program Testing

Program testing is a proper assessment technique in which software requirements, design, or code are examined in detail by a person or group other than the author to perceive faults, violations of development standards, and other problems. Generally, programmers use the phase: desk-checking, translating and debugging to perform proper validation and testing of the program.

- **Desk-checking**: Trace the program code to find out any error that might be there. It is same as proof reading and may expose several errors.
- **Translating**: A compiler is a translator and has in built capabilities of detecting errors and produces a listing of them. These are mostly errors due to the incorrect syntax in the use of language.
- **Debugging**: Debugging is the process of locating, analyzing, and correcting suspected errors. These errors (or bugs) are identified during the execution or running of the program. Most of the errors in this phase are due to the logic of the program.

b) Documentation

Program documentation is a detailed explanation of the programming cycle and specific details about the program. Documentation is an on-going process required to supplement human memory and help organize program planning. Documenting a program is also critical to communicate with other who might have an interest in your program. Typical documentation materials include origin and nature of the problem, brief description of the program, logic tools such as flowcharts and pseudo code, and testing results. The comments written in the program code are also an important part of documentation.

The documentation may be divided into two categories: ***Internal and External Documentation.***

I. Internal Documentation

This type of documentation deals with those aspects of programs which are embodied in the syntax of the programming language, and includes:

- significant names used to explain data items and procedures.
- Comments concerning to the function of the program as a whole and of the modules comprising the program.
- Simplicity of the style and format: i.e indentation of related block of instructions, blank lines separating modules.
- Use of symbolic names (variable names) instead of constants.

II. External Documentation

This category includes supporting documentation, which should be maintained in a manual or folder accompanying any program. It is necessary that as soon as changes are made in a program, its external documentation is updated at the same time. Out-of-date documentation can mislead to a maintenance programmer and result in time wastage.

External documentation should include:

- A current listing of the source program.
- Program specification that is, a document defining the purpose and mode of operation of the program.
- Structure diagram showing the hierarchical organization of the modules currently comprising the program.
- Specification of the data being processed, items in reports and external files processed.
- Where applicable, the format of screens used to interact with users.

3.3.3 Program Maintenance

Program maintenance refer to a process that includes all the changes to a program once it is implemented and processing real transactions. It is natural that at some time, maintenance is required to correct errors that were not found during the testing stage. Other times, maintenance is mandatory to make changes that are the result of users' new information requirements.

Summarizing the above program design concept, following are the principles of good programming:

a. The program requirements must be specified with no ambiguity and should be clear and precise. These specifications will be prepared by a systems analyst or group of programmers who are working on the software design. A programmer has the task of converting these specifications into a written program in the form of pseudo code or flow chart.

b. During development of a program, a programmer should keep working papers. He can refer back to these papers later to check what he has done in case:

   i. If there is an error in the program for correction;

   ii. If the user of the program asks for a change in the program or wants to incorporate some more features in the program.

c. The working papers might include a pseudo code, or flowchart (or both).

d. While writing a program, the programmer should try to keep it as short as possible, since this will make more efficient use of storage capacity in the CPU. The program should therefore be logically well- structured.

e. After making a complete program, it should be tested. A programmer should prepared test data and establish whether the program will process the data according to the specifications given by the system analyst.

f. There should be provision for program amendments. One possible way is to leave space in the program instruction numbering sequence for new instruction to be inserted later. For example, instructions might be numbered initially 10, 20, 30, 40, etc instead of 1,2,3,4.

g. A report should be kept of all program errors that are found during testing and the corrections that are made to the program.

h. Every version of a program should be separated so that it is easy to avoid a mix-up about what version of a program should be used.

3.4 PROGRAMMING TECHNIQUES

In recent years, a range of techniques are evolved those attempt to minimize differences in the way programmers' design and develop software. A few of the most commonly used techniques for standardization are described in this section.

Programming techniques:	
Non-structured	**Structured**
It is a straight forward method for programming in a sequential manner. This type of programming does not involve any decision making. It can be used for handling small & simple problems. A general model of these programs is: (a) read a data value (b) compute intermediate result (c) use intermediate result to compute desired answer (d) print the answer (e) stop Non-structured programming frequently uses **GO TO statement** to transfer control from one part of the program to another part.	Professor E. W. Dijkstra (1960) introduces the term Structural Programming. After some years, Italian scientist C. Bohm and G. Jacopini (1966) gave the basic principal that supports this approach. It is often known as "**GOTO-less**" programming, because it is avoided by programmers. Structured programming is standardize technique used for software development. Structured programming was invented to address the shortcomings of non-structuredprogramming that frequently uses GO TO statements.

Using **GO TO** codes in the program, one can transfer control to backward, forward, or anywhere else within the program. The problem is that the connections between parts of the program by using GO TO commands can become quite messy.

The disorganized and sometimes complicated pattern of linkages between parts of the program is also known *spaghetti code*. This type of programming is difficult to understand and debug.

Such a non-structured programming is now viewed as fruitless programming strategy.

To develop good quality software, developers have to cautiously imagine and design the programs. Now a days, software is expected to follow recognized design principles. The customary design standards are **structured programming** and **structured design**.

3.4.1 Structured programming

Structured programming does not use GO TO commands. In structured programming, the program is divided into several basic structures. These structures are called building blocks and makes use of the control structures (*sequence, selection and repetition*), which we have already studied in previous sections. Here, we introduce again for brevity.

(a) Sequence Structure: This module contains program statements one after another. This is a very simple module of Structured Programming. The sequence principle implies that program instructions should execute in the order in which they appear (see Figure 3.3).

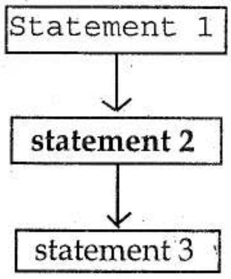


Figure: 3.3 Sequence Structure

(a) Selection or Conditional Structure: The selection rule implies that statements may be executed selectively using IF-THEN and/or IF-THEN-ELSE statements. These statements work in the following way.

IF a condition is met or is true, THEN a specific set of instructions will be executed. If the condition is false, then another set of instructions will be executed.

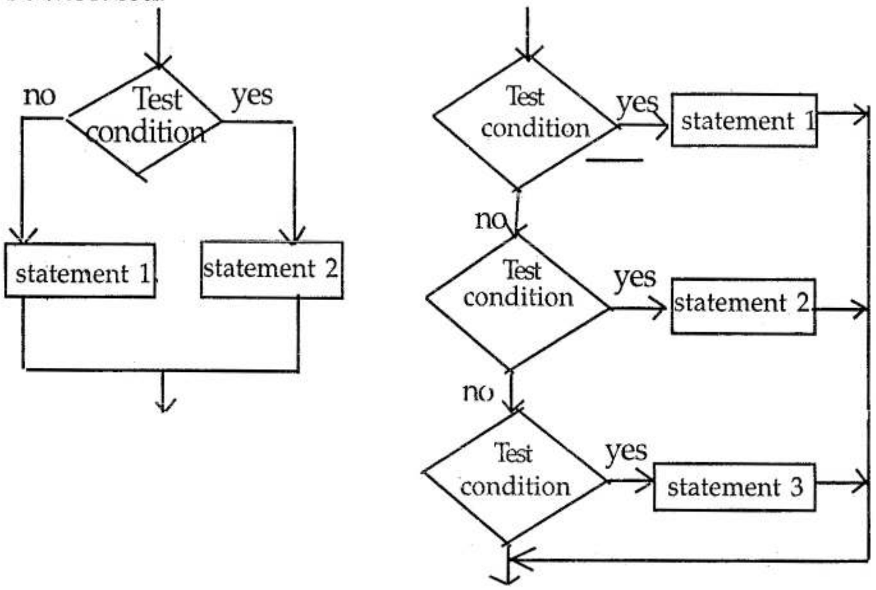


Figure:3.4 Selection Structure

(c) Repetition or Iteration Structure: The ***iteration*** principle indicates that one part of the program can repeat or iterate upto a limited number of times.

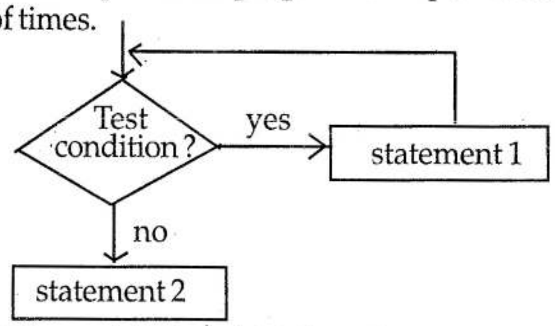


Figure- 3.5 Repetition Structure

3.4.2 STRUCTURED DESIGN

In accordance with structured design principles, a program should be designed from the top-down or bottom-up as a hierarchical chain of modules. A module is a logical way of dividing a program so that each module does one or a small number of related tasks.

a) Modular Programming

The Modular programming breaks a program into subcomponents called modules. Each module consists of an independent or self-contained set of statements. Modules are also known as routines, subroutines, or subprograms or procedures. Each module is designed to do a specific task in the overall program, such as to calculate the total marks of a student in an exam program. In programming language, different names are used for it as:

Programming Languages	Module known as
BASIC, FORTRAN	Subroutine
Pascal	Procedure or function
C, C++, C#	function
Java	Method

Modules are independent and easily manageable. Generally modules of 20 to 50 lines considered as good modules when lines are increased, the controlling of module become complex. Modules are debugged and tested separately and combined to build system.

Advantages:

- Modular programming has the ability to write and test each module independently and in some cases reuse modules in other programs.
- A program consists of multiple modules. In addition, there is a main module (called top module) in the program that executes the other modules.
- You can use top-down or bottom-up approaches in order to design a program consisting of modules.

b) Top-Down Approach

Top-down decomposition is the process of dividing the overall procedure or task into many small **module** or subprogram or function or procedure from top to bottom and then subdivide each component module until the lowest level of detail has been reached. It is called **top-down design** or **top-down decomposition** since we start "at the top" with a general problem and design specific solutions to its sub problems. In order to obtain an effective solution for the main problem, it is advantageous that the sub-problems (subprograms) should be independent from each other. The top module is tested first, and then sub-modules are combined one by one and tested.

This programming approach focuses on developing a software program theoretically before program coding starts. A programming team first creates a diagram that looks similar to an organizational chart with the main module at the top and subordinate modules down below connected by lines with each box representing a program module. The chart shows how modules relate to each other but does not describe the details of the program statements in each module. The structure chart is usually referred to as a Hierarchical Program Organisation (HIPO). So, it is the set of principles that enable a problem to be solved by breaking it down into manageable parts, step-by-step from the overall problem specification.

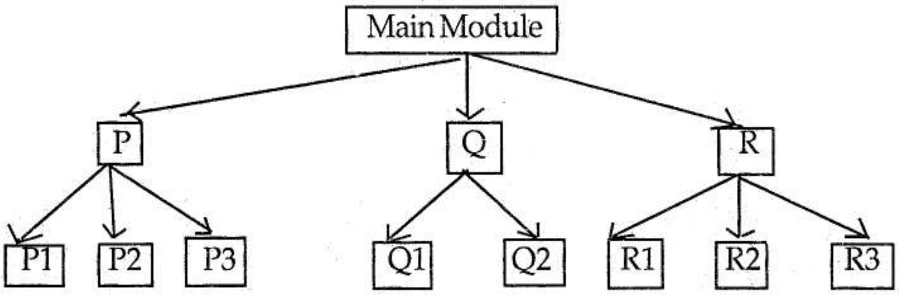


Fig. 3.6 Top down approach

Let us assume that a main module (see Figure 3.6) is divided into sub-program P, Q, and R. The P is divided again into subprogram P1, P2 and P3. The sub-program Q and R is divided into Q1, Q2 and R1, R2, R3 respectively. The solution of main module is obtained from sub program P, Q and R.

c) Bottom-up Approach

In bottom-up approach, already on hand facilities (designs) are used (taken) into consideration as a model for a new design. We take an already existing computer program as a model for our new program. We also try to utilize the existing facilities or design in a way, which gives the out program a better performance. At first, bottom layer modules are designed and tested, second layer modules are designed and combined with bottom layer and combined modules are tested. In this way, designing and testing progressed from bottom to top. In software designing, only pure top down or Bottom up approach is not used. The hybrid type of approach is recommended by many designers in which top down and bottom up, both approaches are utilized.

Check your progress

1. Write pseudcode for a program that will accept names and marks of each student in the class of 50 students, and then calculate the average of all marks. The program should make sure that the entered marks are valid (in the range of 0 and 50). Use the idea of modular programming.
2. Write pseudcode for a program that will read two matrices and then display their sum.
3. The scalar product (also called the inner/dot product) of the two vectors (one-dimensional arrays) A and B with n elements is defined as

   Write a pseudocode, which calls a subprogram with three parameters, A, B, N. The main program should input N and the two arrays A and B. The subprogram should have a name called Product and it should compute the scalar product according to the formula given.

3.5 PROGRAM ERRORS

It is difficult to detect and troubleshoot a software failure. It starts with the problem and the term "error", covers all different kinds of errors, bugs, failures, or faults. A very casual definition of the term "error" is as follows:

*"An **error** is defined as the computation of one or more incorrect results by a computer."*

The above definition of an error has also been used in the definition for debugging. In terms of errors, two kinds can generally be well-known: hardware errors and software errors. The former result from erroneous functions of electronic elements by dust, heat, variations in electrical currents, and others, or have been introduced through corrupted connections and cables or by other faulty components. The hardware errors leads to interruptions of service, which often result in complete breakdowns or system shutdowns. In this course, we are not considering hardware errors, but instead concentrate on the second kind of errors, that is software error.

a. **Software error**

Software errors (known as *logical errors*) are caused by the fact that the software specification is not followed. The program is compiled and executed without errors, but does not generate to produce the required result.

b. **Syntax errors**

These errors occur due to the fact that the syntax of the language is not followed.

c. **Semantic errors**

It refer those errors that are due to an improper use of program statements.

d. **Overflow error**

This type of error occurs when the computer tries to handle a number that is too large for it. It is well known that every computer has a well-defined range of values that it can represent. If during execution of a program it arrives at a number outside this range, it will show an overflow error. Overflow errors are also known as overflow conditions.

e. **Underflow**

It represents a condition that occurs when a computer tries to represent a number that is too small for it (that is, a number too close to zero). Different computer take action to underflow conditions in different ways. Some report an error, while others approximate as best they can and continue processing. For example, if your computer support 4 decimal places of precision and a computation produces the number 0.000006, an underflow condition occurs.

f. Runtime error

Run time error occurs during the execution of a program. For example, if your program tries to access a memory location that is not supposed to access by it, run time error occurs. It also happens due to improper use of pointers, which we shall, studied in C language or by using the division by zero to any number. In contrast, compile-time refers to events that occur while a program is being compiled.

g. linking errors

In some programming languages the in-built library functions or routines available and the user program tries to use these functions. Hence the user includes the files of these functions in the main module or program. If there is any improper use of inclusion or skipping of these files, ten an error occurs which known as linking error.

3.6 SUMMARY

Program design is a very crucial step in software design process; so that it is not so easy to develop a good software. It needs specific *logically related instructions* that the programmer feeds into a computer to solve a particular problem. These instructions are termed as program. A programming language is a set of convention that provides a technique to tell the computer what operations it has to perform. There are two major types of programming language; *low –level* and *high-level* language. Assemblers, compilers and interpreters are system software that translates a source program written by the user in any programming language to an object program which is meaningful to the hardware of the computer. These translators are also referred as language processor since they are used for processing a particular language. *Program designing* is a very sedative phase of *software development cycle.* The objectives of Program designing are: replacement of old system, Organizational requirement, to increase efficacy of old system, competition among the industries.

U.P.Rajarshi Tandon Open University, Allahabad

UGCS-102
Problem Solving Through "C"

Block 2

Introduction to the 'C' Programming Language

Block-2 INTRODUCTION

This block will cover the introduction to programming in 'C'. 'C' is a general-purpose programming language with many features like flow control and data structures, and a rich set of operators. 'C' is neither a high level' language, nor a pure low level language. It is not specialized to any particular area of application. But its nonexistence of restrictions and its simplification make it more convenient and effective for many tasks than apparently more powerful languages. The journey of 'C' was started for designing and implementation of UNIX operating system on the DEC PDP-11, by Dennis Ritchie. 'C' is not fixed for any particular hardware or system so, it is easy to write programs that will run without alteration on any machine that supports 'C'.

Programming in 'C' is an incredible advantage in those areas where you may desire to use Assembly Language but would fairly keep it a "simple to write" and "easy to maintain" program. It has been said that a program written in 'C' will pay a premium of a 20% to 50% increase in runtime because no high level language is as dense or as fast as Assembly Language. However, the time saved in coding can be remarkable, making it the most attractive language for many programming responsibilities. It is possible to write a program in 'C', then rewrite a small portion of the code in Assembly Language and approach the execution speed of the same program if it were written utterly in Assembly Language.

There are various reasons for learning 'C'. 'C' is most likely the hottest programming language around. In fact, many of the best-selling Windows applications were written in 'C'. If you are just come to the programming, 'C' is a great first programming language. If you already know a programming language, such as BASIC or Pascal, you'll find 'C' a praiseworthy addition to your language set.

To understand this and rest subsequent blocks, you don't require any prior knowledge of the 'C' programming language. We will start with the most basic concepts of 'C' and take you up to the highest level of 'C' programming including the concepts of pointers, structures, and dynamic allocation. To fully comprehend these concepts, it will take a good bit of time and work on your part because they are not particularly easy to clutch, but they are very powerful tools. Enough said about that, you will see their power when we get there; just don't allow yourself to worry about them yet.

UNIT-1 Introduction

Structure

1.0 INTRODUCTION

In this unit, we provide basic features that are common to all 'C' programs. The development history of 'C'language is introduced so that you will familiar with 'C'language. The structure of 'C' program is explained that will help to understand the basic terminologies and functions used in 'C' language. This Unit also describes how to create your first 'C' program and how to execute it under different compilers. It is important that you try to understand basic concept of programming which we have already studied in previous block. The 'C' language is not very difficult but rather it is very user friendly and by repeatedly practicing it will make you master in the 'C' programming.

1.1 OBJECTIVES

After going through this unit, you should be able to:

- explain the need of 'C' programming language;
- explain structure of 'C' program;
- Create and execute simple 'C' programs;
- Declare, define and compute the values of variables via simple arithmetic operation;
- Understand examples of escape sequences and program comments;
- Use the input and output statements;

1.2 PROBLEM SOLVING TECHNIQUES

The ability to solve a problem is developed with time. When you try to solve a smaller and simpler problem, you become more attentive about the steps required for writing solutions of the problems. The two

1.2 HISTORY OF C LANGUAGE

The early development of 'C' occurred at AT&T Bell Labs between 1969 and 1973. It was written by Dennis M. Ritchie. It was named 'C' because its features were derived fromformer language called 'B'. Many of the important ideas of 'C' stem from the language **BCPL**, developed by Martin Richards. The influence of BCPL on 'C' proceeded indirectly through the language **'B'**, which was written by Ken Thompson in 1970 at Bell Labs, for the first UNIX system on a **DEC** PDP-7.

The origin of 'C' is closely tied to the development of the Unix operating system, originally implemented in assembly language on a PDP-7 by Ritchie and Thompson. Eventually they decided to port the operating system to a PDP-11. Language **B**'s inability to take advantage of some of the PDP-11's features, especially byte address ability, led to the development of an early version of 'C'. Thereafter, Dennis M. Ritchie rewrites the Unix operating system in 'C' language. The original PDP-11 version of the Unix system was developed in assembly language. By 1973, with the addition of ***struct*** types, the 'C' language had become powerful enough that most of the Unix kernel was rewritten in 'C'. This was one of the first operating system kernels implemented in a language other than assembly. The important thing which was considered for 'C' language is:

"BCPL and **B** are ype less" languages whereas 'C' provides a variety of data types."

Because of powerful and dominant features of 'C', its use quickly spread beyond Bell Labs. In the late 70's 'C' replaces well-known languages of that time like PL/I, ALGOL etc.Everywhere programmers began to write all sorts of programs in C language. Later on different organizations write their own versions of 'C' with a slight difference. This causes a serious problem for system developers. To resolve this problem, the American National Standards Institute (ANSI)formed a committee in 1983 to establish the standard definition of 'C'. This committee approved a version of 'C' in 1989 which is known as **ANSI 'C'**. With littledifferences, every modern C compiler has the ability to staywith this standard. ANSI 'C' was then approved by the International Standards Organization (ISO) in 1990. The development of 'C' in its final form can seen from Figure 1.1.

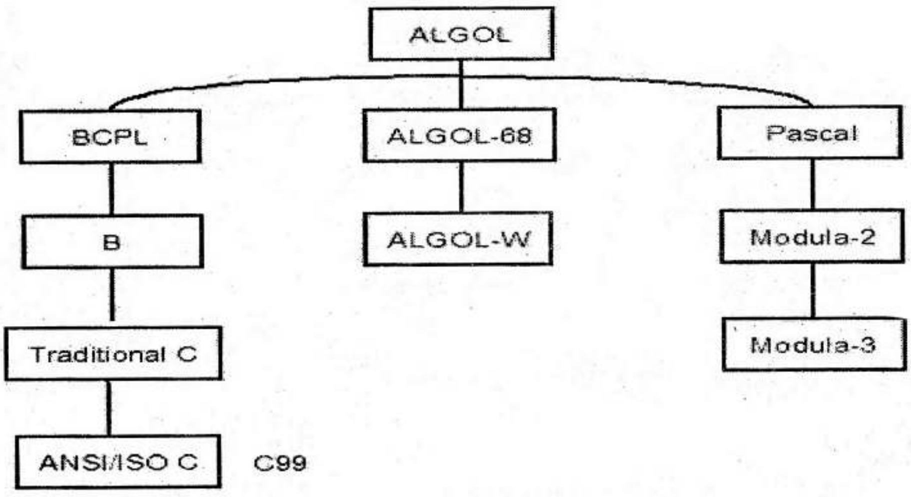


Figure 1.1: Taxonomy of C Language

In contrast to FORTRAN 77 languages, 'C' source code is free-form which allows random use of whitespace to format code, rather than column-based or text-line-based restrictions.

'C' source files contain declarations and function definitions. Function definitions contain declarations and statements. Declarations either define new types using keywords such as*struct, union, and enum,* or assign types to and possibly reserve storage for new variables, usually by writing the type followed by the variable name. Keywords such as ***char*** and ***int*** specify built-in types. Sections of code are enclosed in braces ({ and }, sometimes called "curly brackets") to limit the scope of declarations and to act as a single statement for control structures.

'C' is often used for "system programming", including implementation of operating systems and embedded system applications, due to a combination of desirable characteristics such as code portability and efficiency, ability to access specific hardware addresses, ability to match externally imposed data access requirements, and low run-time demand on system resources.

'C' is a structured programming language. It is considered as a high-level language because it allows the programmer to concentrate on the problem at hand and not worry about the machine that the program will be using. While many high languages claim to be machine independent, 'C' is one of the closest to achieving that goal. This is the another reason that it is used by software developers whose applications have to run on many different hardware platforms.

1.3 STRUCTURE OF A 'C' PROGRAM

It's time to write your first 'C' program! This section will take you through all the basic parts of a 'C' program so that, you will be able to write it. The structure of a 'C' program is shown in Figure 3.2. We will discuss all the components of this structure in detail in the subsequent sections. Right now, this structure is just shown to tell you that almost all 'C' programs follow this.

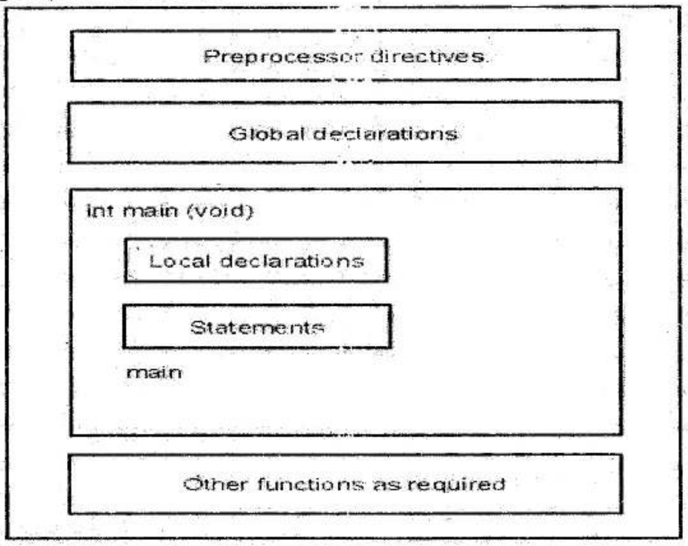


Figure 1.2 Structure of a C program

Let's write our first 'C' program.

```
/* Program 1.1 */
main()
{
}
```

It is suggested tl at you follow the structure of a 'C' program through your first program. Now, we explain each statement of program1.1:

Anything that is written with in the backslash-asterisk pair /* */ are treated as comment. Comments are very useful for debugging and identification of the corresponding task. So **Program1.1** is a **comment** here.

Next, to the comment, there is a word **main**, which is very important, and must appear **once**, and **only once** in every 'C' program. Programs in 'C' consist of **functions**, one of which **must** be **main()**.This is the point where execution of a 'C' program begins. Then other functions may be "invoked". A function is a sub-program that contains **instructions** or **statements** to perform a specific task on its **variables**. When its instructions have been executed, the function returns control to the calling program, to which it may optionally be made to return the results of its computations. Because **main()** is also a function, it has to return control back to the operating system after termination of the program. We will see later that main() does not have to be the first statement in the program but it must exist as the entry point. Following the **main** program name is a **pair of parentheses ()** which are an indication to the compiler that it is a **function**.

The two curly brackets in lines 3 and 4, properly called **braces**, are used to define the **limits of the program** itself. The *local declarations* and *actual program statements* between the two braces and in this case, there are no statements. So your first program does absolutely nothing.

This program can be compiled and you can run it but since it has no executable statements, it does nothing. It is of course a valid 'C' program. When you compile this program, you may get a warning. The warning displays the message that *"main should return a value"*. So either you ignore the warning or modify the program1.1 so that warning message disappears after next compilation.

The modified version of program1.1 is given as in program1.2

```
/* Program 1.2 */
int main()
{
return 0;
}
```

This modified program must compile on any good 'C' compiler since it obeys the ANSI-C standard. In ANSI 'C' standard, by default, main function always returns an integer value. Hence this is the reason to add return 0 in the main function. The return statement must be the last statement in a main function.

Note: In 'C' language, any amount of whitespace is allowed.

Now take a look to another example (**Program 1.3**) which shows that how an output can appear on the screen after the execution of a 'C' program.

```
/* Program 1.3 */
#include <stdio.h>
void main(void) {
        printf("My first C program.\n");
}
```

The first line of Program1.3 is read as *"Hash-include"* stdio.h and written as

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

It is one place where whitespace *is* important. The complete term must be on **one line**, and there can be nothing else on the line. They are also not terminated by a semi colon.

The first line represents a preprocessor directive. The 'C' preprocessor is a program that processes our source program before it is passed to the compiler. In this unit we will not describe more about preprocessor directives, instead we concentrate what exactly is its purpose. The C function used in the example for writing our output by using a function **printf( ),**which is defined in the **stdio.h**library which is included in the program by the statement **#include<stdio.h>.**This statement tellsthe compiler:

- to stop reading our C source file
- first, read the file stdio.h
- then go back to reading our source file

stdio.h is a file that has details of all the input/output functions available in the C library and commonly known as header file.

The word **void***means empty, nothing, none*. When it appears before **main(),** it indicates **no** value will return to the main function. When it appears within the parentheses of a function, in this case;**main()** function, it indicates the compiler that main has no arguments on which it works.

The 'C' library contains many useful functions. One of them is a function **printf( )** statement.

printf("My first C program.\n");

This function outputs the message on the computer's display. A function, including **main(),** may optionally have **arguments,** which are listed in the parentheses following the function name. These are the values of the **parameters** in terms of which the function is defined, and are passed to it by the calling program. In the above example, **main()** has no arguments. The function**printf()** has one argument.It is the bunch of characters enclosed in double quotes. Character sequences between double quotes are called *strings*. The double quotes " do not appear in the output.So the string which is displayed in your monitor screen is

My first C program.

The \n inside the quotes causes **printf** skip to next line.

'C' uses **semicolons** to mark the **end of an expression**or **terminate a statement**, so that, the expression becomes a statement.

Upto this end, we have seen that a 'C' program starts execution from **main()** function and main() can call another function **printf()** which is available in standard C library file **stdio.h.**

We have almost explained all the basic components of 'C' structure. The only thing remains is the explanation of curly braces {.....}.The body of program1.3 is enclosed in braces{}; a pair of braces defines a **block.** A 'C' program may consist of several blocks, which may include another block, and so on. Like Pascal, 'C' is also a **block structured language.** Although left and right braces which mark a block may be placed anywhere on the line, for convenience of reading and debugging they are generally vertically aligned in the same column. Following **program1.4** shows an elegant way to indent your program.

```
/* Program 1.4 illustrates proper use of  indentation */
#include <stdio.h>
int main(void)
{
printf("This is the first level block.\n");
{
        printf("This is the second level block.\n");
        {
                printf("This is the third level block.\n");
                {
                :       printf("This is the fourth level
                :       block.\n");
                }


         }


}
return 0;
}
```

It is noted that in program1.4, each left brace is balanced by a corresponding right brace.

Comments in C language are intended to help people understand your programs, and should be written in a way that its logic becomes transparent. They may be placed wherever the syntax allows whitespace characters: blanks, tabs or newlines. Anything that is written as comment is ignored by a compiler. The compiler **replaces** all the comment with a **single space.**

Block comments in 'C' program are included between /* */ pair. They start with thesequence /* and end with the sequence */.

Example 1:

```
/* this is a block comment
That covers two lines.   */
```

Example 2:

```
/*
It is an example of Block comment.
Here you can add detail about your program.
Many programmers like to add asterisks at
```

the beginning of each line to clearly mark the comment.

*/

'C' language also supports single line comment. The single line comments starts with double forward slash, //.

Example 3:

// it is an example of single line comment.

You must be careful with comments. Don't forgetto put the final */. If you forget */, it means the compiler ignores a lot of your code.

Note: The C language **does not** support **nesting of comments,** so don't embed comments within comments.

Writing in this way

```
/* ========= /* ======= */ =========== */
```

is an example of an **invalid** block comment.

Check your progress 1

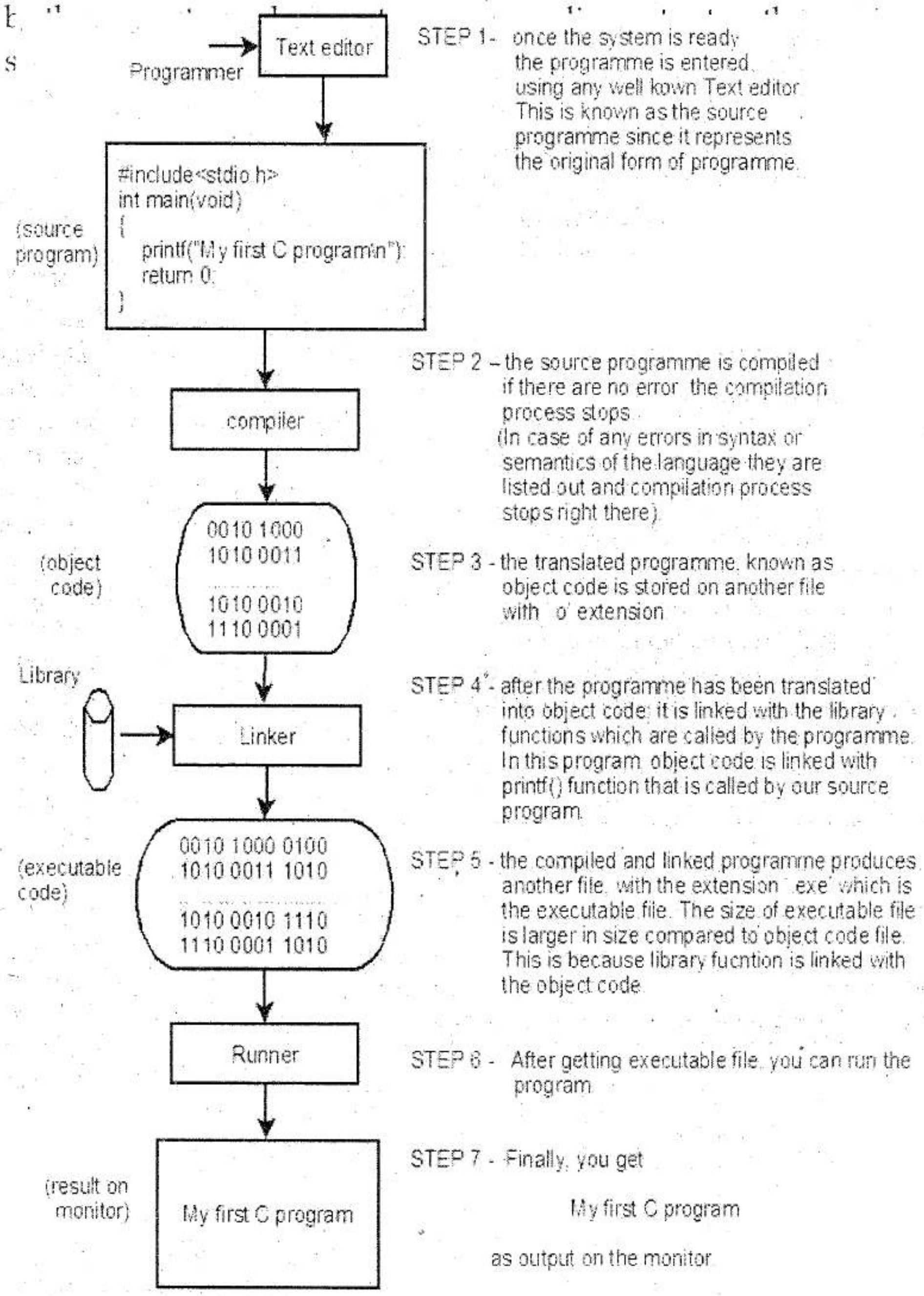
1. Write a program to display your nick name on the monitor.
2. Modify the program to display your address and phone number on separate lines by adding two additional **printf()** statements.

1.4 CREATING AND EXECUTING A 'C' PROGRAM

In this section, we explain the process for converting a program written in'C' into machine language. The procedure (See Figure 1.3) is presented in a straight forward manner but you should be aware of that these steps are repeated many times during development to correct errors and make improvements to the code.

Note: There are many compilers and environments available for writing, compiling, linking and executing a 'C' program. If you run a PC under DOS, you can use **Turbo 'C', Borland 'C', Lattice 'C', Microsoft 'C', Quick 'C', Aztec 'C', Lightspeed 'C'** and many more. These range from the simplistic to thesupportive. If you are using UNIX or LINUX operating system, then command and other features will differ. The most simple systems require the programmer to use a text editor toenter all the code, the code is written to an appropriately name file, compiled, thenlinked to necessary libraries and execution of the code can then follow. The usercorrects mistakes by returning to the editor. More supportive environments tend to beintegrated, with some automatic generation of code; the available libraries may be very extensive.

The Figure 1.3 explain how to create, compile and execute a 'C' program. The source program filename must have **.c** extension to indicate the compiler that it is a 'C' source program. The successful compilation process generates an object file with **.o** extension and executable file with **.exe** extension. This is the **.exe** file which is executed



```
#include<stdio.h>
int main(void)
{
   printf("My first C program\n");
   return 0;
}
```

STEP 1- once the system is ready the programme is entered using any well kown Text editor This is known as the source programme since it represents the original form of programme.

STEP 2 – the source programme is compiled if there are no error the compilation process stops (In case of any errors in syntax or semantics of the language they are listed out and compilation process stops right there)

STEP 3 - the translated programme, known as object code is stored on another file with 'o' extension

STEP 4 - after the programme has been translated into object code; it is linked with the library functions which are called by the programme. In this program object code is linked with printf() function that is called by our source program.

STEP 5 - the compiled and linked programme produces another file, with the extension 'exe' which is the executable file. The size of executable file is larger in size compared to object code file. This is because library fucntion is linked with the object code

STEP 6 - After getting executable file, you can run the program.

STEP 7 - Finally, you get

My first C program

as output on the monitor.

Figure 1.3: Creating and Executing a 'C' program

1.5 ESCAPE SEQUENCE CHARACTERS

Escape Sequence Characters are also known as **Backslash Character Constants.** 'C' supports some special escape sequence characters that are used to do special tasks. Character combinations consisting of backslash \ followed by letter is called **escape sequence.** An escape sequence may be used to represent a single quote as a character constant.

The \n is an example of an escape sequence. It is used to print the newline character. In 'C', it is not possible to use **Enter key** to go next line, so how to start a new line. This can be done with the help of \n character which serve the purpose of 'Enter key'. Putting **\n** in the printf statement

```
printf("My \n first \n C \n program.\n");
```

tells the compiler that after displaying the first string, it goes to second line, displaying second string and so on until no more \n characters are encountered.

So the output of above printf statement is looks like:

```
My
first
C
program
```

If a string does not contain a \n at its end, the next string to be printed will begin in the same line as the last, at the current position of the cursor, i.e. alongside the first string. So the output of two consecutive printf() statements

```
printf("This is My first C program.");
printf("It is very easy to write C program");
```

is:

```
This is My first C program.It is very easy to write C
program
```

Some of the escape sequence characters are as follow:

Characterconstant	meaning	action
\n	new line (line break)	Go to next line.
\b	backspace	Takes the cursor back to one position left.
\t	horizontal tab	Moves cursor 1 tab right.
\f	form feed	It forces the printer to eject the current page and to continue printing at the top of another.
\a	alert (alerts a bell)	Ring a bell.
\r	carriage return	sent the print head back to the start of the line.
\v	vertical tab	Vertical tab advances the page by several lines without a carriage return.
\?	question mark	Displays ?
\'	single quote	Displays '
\"	double quote	Displays "
\\	backslash	Displays \
\0	null	To indicate end of string

\x*nn*	Hexadecimal character code *nn*, where n={0,1....F}	Displays Hexadecimal value nn
\o*nn*	Octal character code *nn*, where n={0,..7}	Displays Octal value nn
nn	Octal character code *nn* where n={0,..7}	Displays Octal value nn

In C, **""** is the **null string**. The null string is not "empty", as one might thought; it contains a single character, the **null character**, ASCII value 0 (in which all bits are set to zero).

Check your progress 2

1. Give the output of the following program/* **Program 1.4** */ #include<stdio.h> main(){printf("This\nis\a\nthe \n\nway of writing, \"What about you?\"\n");}

2. Write a C program that generates the following output: /
* It is a one line comment */

3. Execute Program 1.5 below and obtain the answer to the question in its **printf():/* Program 1.5 */**

```
#include <stdio.h>
int main (void) {
printf ("bell alert          \\a   %d\n" , '\a');
printf ("backspace           \\b   %d\n" , '\b');
printf ("horizontal tab      \\t   %d\n" , '\t');
printf ("newline             \\n   %d\n"  , '\n');
printf ("vertical tab        \\v   %d\n"  , '\v');
printf ("formfeed            \\f   %d\n"  , '\f');
printf ("carriage return     \\r   %d\n"  , '\r');
printf ("double quote  \"    \\\"  %d\n", '\"');
printf ("single quote  \'    \\\'  %d\n", '\'');
printf ("question mark  ?    \\?   %d\n"  , '\?');
printf ("backslash      \\   \\\\  %d\n", '\\');
return 0;
}
```

4. Give the output of the following program

```
/*             Program              1.6            */
#include                                   <stdio.h>
main(void)
{
printf("\aEnter your desired monthly salary:");/* 1 */
printf(" $_______\b\b\b\b\b\b\b");          /* 2 */
return0;

}
```

5. Give the output of the following program

```
/* Program 1.7 */
#include <stdio.h>
main(void)
{
        printf("What is your name: \t");
        printf("\"My name is Ramesh\"");
return 0;
}
```

6. Point out the errors in the following program:

```
#include  <stdio.h>
Main(void)
{
print("\p This is wrong or right?");
}
```

1.6 EXAMPLE PROGRAMS

In this section we present few programs those involve concepts which have not so far been discussed. They will be covered later in this Block at appropriate place. Many of the illustrated programs in this section are self explanatory and it is advised to you to go through these programs and understand the essence of each statement. You can also try to write, compile and execute them.

```
/* Program 1.8 */
#inlcude<stdio.h>
voidmain()
{       /* illustration of elementary operations with small integers
        In C, int represent small integers */
        int a=7, b =9, c;       /*
                                a, b, c are int variables
                                a is 7, b is 9, c does not have a value yet
                                */
        printf("a=%d, b=%d\n", a,b);
                                /*
                                each %d prints a decimal number
                                */
        c = a + b;              /*
                                now c is addition of a and b
                                */
        printf("c=a+b=%d \n",c);
        c = a * b;              /*
                        the * means ' multiplied by'
                        */
        printf("c=a*b=%d \n",c);
        c = a / b;              /*
                        int a is divided by another int b
```

```
                              */
       printf("c=a/b=%d \n",c);
       c = a % b;                /*
                              % stands for remainder. So c gets the
                              remainder value when a is divided by b
                              */
       printf("c=a%%b=%d \n",c);/* %% is used to print % sign */
       c = a % b;
       printf("c=a%%b=%d \n",c);
}
/* Program 1.9 */
#include<stdio.h>
voidmain()
{      /* read values from the keyboard, and see how scanf() works */
       int a, b, c;
       printf("Enter a value for a. Type a small integer, press <Enter>");
       scanf("%d", &a);       /*
                      notice the ampersand "&", just before a
                      */
       printf("Enter a value for b. Type a small integer, press <Enter>");
       scanf("%d", &b);
       c = a * b;
       printf("c = a* b = %d\n", c);
}
/* Program 1.10 */
#include<stdio.h>
voidmain()
{      /* program to compute greatest among two numbers
       See how if statement of C language works */

       int a, b;
       printf("Enter a value for a");
       scanf("%d", &a);
       printf("Enter a value for b");
       scanf("%d", &b);
       if( a > b)              /* Is a greater than b?. */
              printf("a is greater than b\n");
       else                    /* else, if it's not, b is greater than a. */
              printf("b is greater than a\n");
}
```

Here, we show the basic difference between ANSI C and K&R style C

ANSI C style program	K&R style C program
/* Program 1.11 */#	/* Program 1.12 */
inlcude<stdio.h>	#inlcude<stdio.h>
int addtwo(int a, int b); /* a function that adds two ints */	int addtwo(); /* see difference here */
main()	main()

```
{ int a, b, sum;
  printf("Enter a value for a");.
  scanf("%d", &a);
  printf("Enter a value for b");
  scanf("%d", &b);
  sum=adtwo(a,b); /* transfer control
                  to addtwo()
                   function, with
                   arguments a, b
                   */
  printf("addtwo( ) tells us their
  sum = %d\n", sum);
  }
 int addtwo(int p, int q)
 {
 printf("i am calling from addtwo()
  and ");
 printf("recently i receive two
 numbers ");
 printf("%d and %d", p, q);
 printf("...their sum is ... ");
 return (p+q);
 }
```

```
{
int a, b, sum;
printf("Enter a value for a");
scanf("%d", &a);
printf("Enter a value for b");
scanf("%d", &b);
sum=adtwo(a,b); /* transfer control
                  to addtwo()
                  function, with
                  arguments a, b
                   */
printf("addtwo( ) tells us their
sum = %d\n",
 sum);
 }
 int addtwo(p, q)/* Note the difference
between ANSI C and
K&R C style */
 int p, q;
  {
printf("i am calling from addtwo() and
");
printf("recently i receive two numbers
");
printf("%d and %d", p, q);
printf("...their sum is ... ");
return (p+q);
}
```

1.7 SUMMARY

In this Unit, we presented a historical development of 'C' programming language, explaining the advantages it have to both system and application developers. We also showed how to write a simple 'C' programs. We have also undergone some important terms like, **main()**, **printf()**, function, block, arguments, parameters, statements, semicolon, comments, null string, null character, escape sequences and preprocessing directive #**include<stdio.h>**.

UNIT 2: DATA TYPES IN 'C'

Structure

2.0 INTRODUCTION

Any computer program has two entities to consider, the data, and the program. They are highly reliant on one another and careful planning of both will lead to a well planned and well written program. Unfortunately, it is not possible to study either completely without a good working knowledge of the other. The 'C' language is rich and supports many basic built in data types. The **typed** language supports pre defined **types** of data or variables in their program. They also pose restrictions on the kinds of operations performed on a particular type. Almost all languages are differ one from another with respect to the degree of typing. In contrast to Pascal, which is a **strongly typed** language, 'C' is **loosely typed** language. Pascal does not permit doubtful operations such as addition of a character value to a real value. Another benefit of typing is that it helps the compiler to allot appropriate amount of memory for each program variable; i.e. one byte for a character, two for an integer, four for real variables etc. But 'C' provides more degree of flexibility to the programmers to write code of greater reliability and robustness.

2.1 OBJECTIVES

After working on this unit, you should be able to:

- explain the meaning of tokens and its various categories.
- explain identifiers, and how to construct a valid identifier.
- construct a valid constant and variables.
- Understand various keywords of C language.

- Understand the basic data types, **int**, **float**, **double**, **char**; and range modifiers; **short**, **long**, **signed** and **unsigned**.
- Declare and define the variable

2.2 CHARACTER SET OF 'C' LANGUAGE

The characters can be used to form words, numbers, and expressions. The characters or character set in 'C' are grouped into the following categories, as listed in Table 1:

Table 1: Character set of 'C'

Category	Characters		Description
Letter	A .... Z		All upper case letters
	a............... z		All lower case letters:
Digits	0,1,2,3,4,5,6,7,8,9		All decimal number
Special characters			
Characters	Description	Characters	Description
,	Comma	&	Ampersand
.	Period	^	Caret
;	Semi-colon	*	Asterisk
:	Colon	-	Minus
?	Question mark	+	Plus
'	Apostrophe	<	Less than
"	Quotation mark	>	Greater than
!	Exclamation mark	(	Left parentheses
|	Vertical bar	)	Right parantheses
/	Slash	[	Left bracket
\	Backslash	]	Right bracket
~	Tiles	{	Left brace
_	Underscore	}	Right brace
$	Dollar	%	Percent
#	Number sign		
White space	Space	Form-feed	Carriage return
	horizontal-tab	New line	

2.3 TRIGRAPH CHARCTERS

The source character set of 'C' source programs is contained within seven-bit ASCII character set, but is a superset of the ISO 646-1983 Invariant Code Set (K & R).

Trigraphs are a sequence of three characters starting with two consecutive question marks(**??**) followed by ***another character*** and allow the compiler to replace with their **corresponding punctuation characters**. Trigraph sequences are used in 'C' program files for some keyboards which do not support some characters mentioned in Table 2(**Special Characters**).

The following Table 2 shows the list of **trigraph sequences** followed by an example.

Table 2: Trigraph sequences and their equivalent punctuation character

Trigraph sequence	Punctuation character	Description
??=	#	Number sign
??(	[	Left bracket
??)	]	Right bracket
??<	{	Left brace
??>	}	Right brace
??!	|	vertical bar
??/	\	backslash
??′	^	Caret
??-	~	tilde

Let us consider the following program, try to print th string**??<character??>**with**printf()** statement

```
#include <stdio.h>
main()
{
printf( "??<Character ??>" );
}
```

The string which is printed on your monitor screen is:{ **Character** } because **??<**and **??>** is **trigraph sequences** that are replaced with the "{" and "}" character.

2.4 TOKENS

The 'C'program is split into tokens separated by white-space characters. The program replaces all comments by a single space.There are six categories of tokens: *identifiers, keywords, constants, string literals, operators,* andother *separators*. Blanks, horizontal and vertical tabs, newlines, form-feeds and comments which we have already discussed are ignored except as they separate tokens. The purpose of splitting a program into tokens is to tell the compiler that which token belong to which category.

Let us consider a 'C' statement.

diff = 7 - 4;

Then, this statement is split into following tokens shown in Table 3. The Table 3 also describes its category.

Table 3

Token	Token type (Category)
diff	Identifier
=	Assignment operator
3	Integer constant
+	Addition operator
2	Integer constant
;	End of statement (separator)

Exercise 2.2: Given below are some identifiers. Specify which of them are valid identifiers and which are invalid and why?

(a) rname	**(b)** SIZE_ARRAY	**(c)** 12_5
(d) ABC	**(e)** $_IS_MINE	**(f)** Var2
(g) Size of array	**(h)** Is2+2=4?	**(i)**_Is-valid

2.6 KEYWORDS

The **keywords**or **reserved words**can be considered as **identifiers** those are **reserved** to have **specialmeaning**.These words are not allowed to be used by programmer for their own purpose. There are **32** words defined as **keywords** in C. They are always written in lower case. *For example***int** is used for representing **integer**.The following identifiers are reserved as keywords in 'C' language, and should not use otherwise

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

2.7 CONSTANTS

Constant in 'C' refer to data values that **cannot** be changed during the execution of a program. Like variables, constants also have a data type. We shall discuss the meaning of data type in subsequent sections. C supports several types of constants (see Figure 1.4). The constants are broadly divided into two categories: **numeric** and **character** constants.

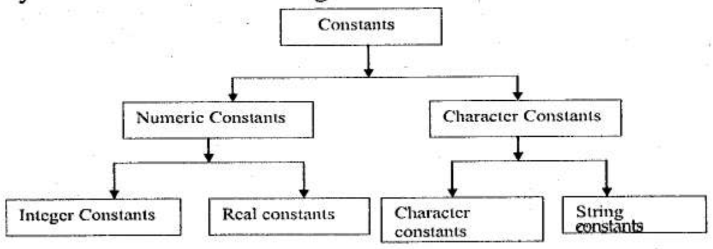


Figure 1.4: Types of constants

2.7.1 Integer Constants

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

- o Decimal integers: It consist with set of unique digits i.e from0 through 9,preceded by an optional -or + sign. For examples: 123, 23468, 0, -78, +39 etc.
- o Octal integer constant: consists of any combination of digits from the digit 0to 7, with a leading 0. For example: 037, 0, 0435,0551.
- o Hexadecimal integer constant:A sequence of digits preceded by 0x or 0X is considered as hexadecimal integer. They may also include alphabets 'A' through 'F' or 'a' through 'f' which indicates 10 through 15. For example: 0x2, 0x9f, 0Xcbd etc.

There are following rules for constructing the Integer Constants:

1. An integer constant must have at least one digit.
2. It must not have a decimal point.
3. it could be either positive or negative.
4. if no sign precedes an integer constant it is assumed to be positive.
5. No commas or blanks are allowed within an integer constant.
6. The largest integer value that can be stored is machine-dependent.
7. The allowable range for integer constants is -32768 to +32767. It is 32767 on 16 bit machines and 2,147,483,647on32-bit machines.

Example of some valid numeric constants

Constant	Type	Constant	type
245	Decimal integer	035	Octal integer
-68	Decimal integer	038	Octal integer
+123	Decimal integer	0x234	Hexadecimal integer
0	Decimal integer	0X24B	Hexadecimal integer

Example of some invalid numeric constants

Invalid Constant	Remark
12 45	White space is not allowed
-5,68	Comma (,) is not allowed
$123	Illegal use of $

2.7.2 Real constant

The real constants are often called Floating Point constants. The real constants could be written in two forms: **Fractional form** and **Exponential form**. The rules for constructing Real constants expressed in Fractional form are as follows:

1. A real constant must have at least one digit.
2. It must have a decimal point.
3. It could be either positive or negative.
4. Default sign is positive.
5. No commas or blanks are allowed within a real constant.

Anexample for real constant in fractional form is:426.0,-32.76,-48.236

The exponential (or scientific) form of representation of real constants is usually used if the value of the constant is either too small or too large. In exponential form of representation, the real constant is represented in two parts. The part appearing before 'e'; is called **mantissa**, where as the part following 'e' is called **exponent**.

Rules for constructing Real constants expressed in Exponential form are as follows:

(i) mantissa part and the exponential part should be separated by a letter '**e**'
(ii) The mantissa part may have a positive sign.
(iii) Default sign of mantissa part is positive.
(iv) The exponent have at least one digit which can be eitherpositive or negative integer. Default sign is positive.
(v) Range of real constants expressed in exponential form is -3.4e38 to3.4e38.

Example of some valid real constants

0.003	-.71
+23.123	0.45e3
-0.536	0.78e-3
336.0	1.23e6
.358	-5.24e-2

2.7.3 Character Constant

A character constant is a **single alphabet**, a **single digit** or a **single special symbol** enclosed within single inverted commas. The escape sequences which have discussed in UNIT 1 are also an example of character constant. For Example: 'A' is a valid character constant whereas 'aA' is not.

The rules for constructing Character Constantsare as follows:

(a) The maximum length of a character constant can be 1 character.

(b) Since each character constant represent an integer value so, it is also possible to perform arithmetic operation on character constant.

Each character constant has an ASCII value associated with it. For example, the following statements will print 66 and B respectively.

printf("%d", 'B');// **%d specifier is used to print decimal value (i.e. ASCII value of B)**

printf("%c", '66'); // **%c specifier is used to print character (i.e. ASCII character of 66)**

Some **examples** of character constant are: '$' ,'A','5', '+'.

1.7.4 String Constant

A **string constant**or **string literal**is any sequence of characters enclosed in pair of double quotes, such as **"abc"**. The characters may be letters, number, special characters and blank space.String constants are always treated differently from character constants. Forexample 'a' and "a" are not the same.Some examples of string constant are: "hello!","1987", "Well Done", "?......I ?" ,"X". In C, all string constant are terminated by '\0' (or NULL string).

Check your progress 1

1. First character in any variable name must always be an a ____________________
2. C variables are case ____________
3. A character variable can store ________________ character at a time.
4. Point out which of the following C constants are invalid:(a) 21.34 (b) 023 (c) 0xtc40 (d) -23 .4
5. Give the output for the program
```
main()
{
printf("%d %d %d %d", 62, 062, 0x62, 0x62);}
```
6. Give the output for the program
```
main()
{
char letter = 'T';      printf("%d", letter);
}
```
7. Execute and predict the output for the program
```
main()
{
        char num1 = 't';
        char num2 = 'a';
        int add;
add = num1 + num2;
printf("the sum of %d and %d is = %d", num1, num2,
add);
printf("the sum of %c and %c is = %d", num1, num2,
add);}
```

8. Give the output for the program

```
main()
{
printf("??(\n\nnn\n\n??));
}
```

9. Write a program to read the distance in decimal form (like 21.34 Km) and print the output in meters (like 21340 meter). [**Hint:** use **scanf**()to read values from keyboard]

10. Write a program to interchange the values of a and b without using third variable.

2.8 DATA TYPES

Each program requires a particular type of data for displaying a meaningful result. These particular kinds of data are known as data type. A type defines a set of values and a set of operations that can be applied on those values.

'C' also has all the standard data types as in any high level language. C has **int, short, long, char, float, double**basic data types. 'C' has no Boolean data type or string type. 'C' has no Boolean type but **0** can be used for **false** and **1** can be used as**True**.

The memory necessities of each type will differ on different computers, and in fact can vary with different compilers on the same processor. The range of data types available allows the programmer to select the type appropriate to the needs of the application as well as the machine.

The Table 4 shows the basic built-in data types available in 'C' language along with their possible range of values and sizein bytes occupied by that data type variable in memory. It is noted that the size taken by each data type is **machine dependent**. For most compilers as the IBM machine **int**data types occupied in 2 bytes, and are restricted to the range [-32768, 32767]. Compare this with the Macintosh machine, where **ints** are 4 byte signed integers in the range [-2147483648, 2147483647].

Note: The use of the standard headers **<limits.h>**and **<float.h>**can help in fixing sizes when trying to write portable programs.

Table 4: Basic built in data types in C

Note: Size depends upon machine architecture

Data type	Size (in bytes)	Range
int	2	-32,768 to 32,767
float	4	-3.4e-38 to 3.4e+38
double	8	1.7e-308 to1.7e+308
char	1	-128 to 127

(I) The **int** data type is used to define integer numbers.

```
/* Program 2.1 */
#include<stdio.h>
voidmain()
{
intnumber;
number =25;
}
```

In the above program2.1, **number** is a variable name that has **int** data type. This tells the compiler to allocate **2 bytes** of memory for **number** variable. Now, the range of **number** lies in between -32,768 to 32,767. It means that this variable can hold any values that lie in this interval. If you are trying to assign some more value that is beyond this range, than you will get erroneous results.

Now, it's an exercise for you to check the result for program2.2. What output do you expect and what will you see in your monitor screen!

Exercise:

```
/* Program 2.2 */
#include<stdio.h>
voidmain()
{
int salary;
salary =252341;
printf("Salary is Rs. %d\n", salary);
}
```

Note: The **%d** in **printf** statement is used to print **integer values**.

Execute the program below, with the indicated arithmetic operations, to determine their output.

```
/* Program2.3 */
    #include<stdio.h>
    voidmain()
    {
          int x =30, y = 20, z;
          printf("x = %d\n", x);
          printf("y = %d\n", y);
          z = x – y;
```

```
            printf("Their difference x – y is = %d\n", z);
            z = x * y;
            printf("Their product x * y is = %d\n", z);
            z = x / y;
            printf("Their quotient x / y is = %d\n", z);
        }
```

In particular, determine the values you obtain for the quotient x/ y when:

```
        x = 40, y =20;
        x = 50, y =20;
        x= 60, y =20;
        x=70, y =20;
```

Can you explain your results?

(II) **The float** data type is usec to define floating point numbers.A **float** data type number uses 4 bytes giving a prec ion of 7 digits. So these numbers are known as **single precession numbers**.

```
/* Program 2.4 */
#include<stdio.h>
voidmain()
{
        float ratio;
        ratio =2.5;
        printf("value of ratio is %f", ratio);
}
```

Note: The **%f** in **printf** statement is used to print **float values**.

In program2.4, **ratio** is a variable name that has **float** data type. This tells the compiler to allocate **4 bytes** of memory for **ratio** variable. The range ofvariable **ratio** lies in between -3.4e-38 to 3.4e+38. It means this variable can hold any values that lie in this interval. If you are trying to assign some more value that is beyond this range, then you will get erroneous results.

(III) **The double** data type is used to define large floating point numbers. It takes 8 byte of storage in memory and its range lies from 1.7e-308 to1.7e+308.A double data type number uses 8 bytes giving a precision of 14 digits. These are known as **double precession numbers**and have a much greater range of definition than **floats.** . Remember that **double** type represents the same data type that float represents but with a greater precision. The **precision** means the number of significant digits after the decimal point.

To extend the precision further, we may use **long double** which uses 10 bytes.(**long** is described in next section).

Execute the program2.5 which computes the volume of sphere, whose radius is read from the keyboard.

```
        /* Program 2.5   */
```

```
#define PI 3.1415928          //MACRO definition
#include<stdio.h>
voidmain()
{
        double radius, volume;
        printf("This program computes volume of sphere\n");
        printf("Enter radius of sphere: ");
        scanf("%lf", radius);     //%lf is used to input double values
        printf("Volume of sphere is (4/3) * pi* r * r * r\n");
        volume = (4/3)*pi*radius*radius*radius;
        printf("Volume of sphere is :%lf\n"), volume); //% lf is used to
        output double values.
}
```

Note about Program 2.5

The **%lf** (or **%le** or **%lg**) in scanf() statements are required for the input of double variables.To get output of double variable using printf(), use **%e** (or **%E**), **%f** or **%g** for single precision.

The **#define** also called **MACRO,** it is used to declare constants in C. Like **#include**, **#define** are also preprocessor control lines. The general usage of **#define** is:

#define IDENTIFIER value_to_be_replaced

It is noted that there should be no space between # and **define.** Like **#include**, **#define** are also not terminated by semi-colon. If you place semi-colon there, it will become a part of replacement string and cause syntactical error.

A #define quantity is **not** a variable and its value cannot be modified by an assignment. A major **benefit** of MACRO definition is that if the replacement string has to be changed throughout the program, than it has to be changed **once** in a MACRO definition.

ANSI C also provides the **const** declaration for items whose value should not be change in a program:

const int speed = 2500;

The keyword **const** lets the programmer specify the type explicitly in contrast to the **#define**, where the type is deduced from the definition.

End of Note].

V) The char data type is used to define character data type. It takes byte of storage in memory and its range lies from -128 to 127.

Consider Program2.6 which illustrates **double** and **char** data types.

```
/* Program 2.6 */
#include<stdio.h>
void main()
{
        double ratio;
        ratio =350000;
        char Decision;
        decision ='Y';
        printf("value of ratio is %lf", ratio);
        printf("Your decision is %c", Decision);
}
```

Note: The % **lf** in **printf** statement is used to print **double values**.

The **%c** in **printf** statement is used to print **char values.**

The Program2.6 declares **ratio** and **Decision** as **double** and **char** data type respectively. This tells the compiler to allocate **8** and **1** byte of memory locations to the variables **ratio** and **Decision**respectively.

The Table 4 shows that **char** data type needs 1 byte in memory. Since 1 byte consists of 8 bits, the **char** data type has **2^8 = 256** possible values, which constitute its range. That's why **char** data type has range in between -128 to 127, thus total of 256 possible values. As **int** data type takes 2 bytes (or 16 bits), it has **2^{16} = 65536** possible values. So positive and negative values covered by **int** data type includes from -32,768 to 32,767, thus total of 65536 values. The same argument is also applied for **float** and **double** data types.

It's an obvious question that range of these built in data types are very small. If we are trying to store an integer value that exceeds the range of **int** data type, than what should we do? The answer is C provides us some **range modifiers** to increase the predefined range of data types. This is the next topic for discussion.

Execute and check the output of the Program2.7. What you observe from these outputs?

```
/* Program 2.7 */
#include<stdio.h>
voidmain()
{
        int cost = 42000;
        int item = 12;
        float cost_per_item= cost/item;
        printf(""The cost per item is %f\n", cost_per_item);
        doublelight_speed = 300000000;
        int time = 50;
        float distance;
        distance = light_speed * 50;
        printf("Distance covered by light in %d time is : %f", time,
               distance);
}
```

Note: The **%f** in **printf** statement is used to print **floating values**.

2.8.1 Range modifiers

The data types explained above have the following range modifiers (or qualifiers).

- short
- long
- signed
- unsigned

The **modifiers** define *how much amount of memory is allocated to the variable.*

To alter the amount of memory, use **short** and **long** qualifiers. The qualifiers **short** and **long** apply to **integers**.

For example:

short int qty;

long int distance;

To change the sign, use **signed** and **unsigned** qualifiers. The qualifier **signed** or **unsigned**can be applied to **char** or any **integer**.

Unsigned numbersare always positive or zero and **unsigned char** variables have valuesbetween 0 and 255, while **signed char** have values between -128 and 127 (negative numbers are represented in 2's compliment form.)

ANSI C has the following rules for the range modifiers:

```
short int <=     int     <=     long int
float     <=     double  <=     long double
```

The above rule states that a **'short int'** should assign less than or the same amount of storage as an **'int'** and the **'int'** should be less than or the same amount of storage as an**'long int'**.

! A word of caution: The meaning of these types will vary between compilers, for example on some machines **int** and **short int** will represent the same range. You are advised to check your compiler manual before porting your program from one machine to another; otherwise, you may face an overflow error. [**End of caution**] !

Table 5represents the names of some most commonly used data types and their possible range of values and size in bytes on a 16-bit machine.

Table 5: Size and range of basic data types in C

	Type	Bytes	Range
Integer	**int or signed int**	2	-32768 to 32,767
	unsigned int	2	0 to 65,535
	short int or **signed short int**	1	-128 to 127
	unsigned short int	1	0 to 255
	long int or **unsigned short int**	4	-2,147,483,648 to 2,147,483,647
	unsigned long int	4	0 to 4,249,967,295
Floating point	**float**	4	3.4e -38 to 3.4e+38
	double	8	1.7e-308 to 1.7e+308
	long double	10	3.4e -4992 to 1.1e+4932
Character	**char** or **signed char**	1	-128 to 127
	unsigned char	1	0 to 255

Execute the program and determine its output. Also examine the effect of changing the **%u** format conversion specifiers to **%d** in the printf()s. Can you explain your results?

```
/* Program 2.8 */
#include<stdio.h>
voidmain()
{
        unsigned int hall_seats, tickets_sold, viewers_standing;
        hall_seats = 50000;
        tickets_sold = 60000
        viewers_standing = tickets_sold - hall_seats;
        printf("ticket sold : %u\n", tickets_sold);
        printf("seats availbale : %u\n", hall_seats);
        printf(there could be rush because \n");
        printf( there may be nearly %u standees at the match.\n",
        viewers_standing);
}
```

Note: %**u** is used to output **unsigned int**.

Note: When a long integer constant is assigned a value the letter **L** (or **l**) must be written immediately after the rightmost digit:

long int big_num = 1234567890L;

Execute the program and determine its output. Also examine the effect of changing the **%ld** format conversion specifiers to **%l and %d** in the printf()s. Can you explain your results?

```
/* Program 2.9 */
#include<stdio.h>
voidmain()
{
        long int population_2010 = 1234567890l;
        printf ("the pospulation of country in 2010 \n");
        printf(will exceed %ld if we do\n", population_2010);
        printf("not take concrete steps now\n");
}
```

Note:%ld in printf() output **long decimal ints.**

Note: The **unsigned long** declaration transforms the range of **long int** to the set of 4 byte non –negative integers. **unsigned long** are output by **%lu** format conversion specifier in the **printf().**

Experiment with the following program to see what happens if any of x or y or z exceeds the limit of **ints** for your computer.

```
/* Program 2.10 */
#include<stdio.h>
int main()
{
int x, y, z;
        printf("type value for x\n");
        scanf("%d", &x);
        printf("value of  x is \n", x);
        printf("type value foryx\n");
        scanf("%d", &y);
        printf("value of  y is \n", y);
        z = x + y;
        printf("The sum of x and y is %d", z);
        return 0;
}
```

If you are interested to know about how much storage is allocated to a **data type**, you can use **sizeof** operator which we will discuss in next unit.

The **sizeof**() is a **special operator**which returns number of bytes taken by a data type.

Execute the Program2.11 to check the size of memory taken by various data types.

```
/* Program 2.11 */
#include<stdio.h>
int main()
{
        printf("sizeof(char) == %d\n", sizeof(char));
        printf("sizeof(short) == %d\n", sizeof(short));
        printf("sizeof(int) == %d\n", sizeof(int));
        printf("sizeof(long) == %d\n", sizeof(long));
        printf("sizeof(float) == %d\n", sizeof(float));
        printf("sizeof(double) == %d\n", sizeof(double));
printf("sizeof(long double) = %d\n", sizeof(long double));
printf("sizeof(long long) = %d\n", sizeof(long long));
return 0;
```

2.9 VARIABLES

Suppose you have to store the data, so you require specific memory location. But it is very difficult for us to remember the physical address of stored data. So what we do, we assign some **name** to the **physical address** that contains the data and this name is called **logical address** or **variable name**. It is called **variable** because its value (data value) can be changed throughout the program execution. Which type of value is to be stored is determined by the **type** of data. If data value belongs to some whole numbers, than its type should be **int**. Thus, eachvariable has a specific type,which is known as **variable data type**. This data type tells the computer how muchmemory is occupied by the variable to store the data.

The rules for constructing different types of variable names are as follows:

(1) A variable name may consists of letters, digits and underscore characters.

(2) The first character in the variable name must be an alphabet. Some systems permit underscore as the first character.

(3) No commas or blanks are allowed within a variable name.

(4) No special symbol other than an underscore can be used in a variable name.

(5) Uppercase and lowercase are significant. For example; variable CUST_ID is not same as Cust_Id and cust_Id.

(6) The variable name should not be a 'C'keyword.

Some example of valid variable name:

Cust_Id	Min_Salary	Name_Student
Amount	Height	box

Some example of invalid variable name:

@2	**!Hello**	**13th**	**(address)**

2.9.1 Declaration of variables

Any variable used in the program must be declared before using it in any statement. To accomplish this task, the type declaration statement is used and its syntax is as:

data_type variable_name1, variable_name2,. . . ;

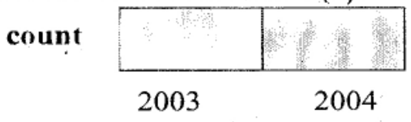
If more than one variables are declared, they are separated by **comma** (,). The declaration statement must be ended by **semi-colon** (;).

For example, the declaration

int count;

tells the following things to the compiler:

- **count** is thea name of variable.
- **count** is **logical address**of variable.
- **count** has **int** data type, so it is allocated 2 bytes of memory on 16 bit machine.
- Suppose **'2003'** is the **physical address** of variable **count**. It is **noted** that this physical address may vary on each time when the statement is executed. Here we have chosen 2003 just for illustration purpose.
- Presently, it has given no value to store. It is **noted** that when we assign nothing to a variable, it automatically takes **any value**. **Do not think** that it has value zero (0).



Thus, following descriptive statement can be made: "***count** is a name of variable, which has **int** data type and it gets 2 byte of memory irrespective of what physical address it have*".

Now, state what you infer from the given declarative statements:

char value;

float result,sum;

double add;

Execute the following program and determine its output on your computer.

```
/* Program2.12 */
#include<stdio.h>
voidmain()
{       int a, b, c ;     /* a, b, c are undefined   */
        int z;
        z= a + b;         /* z is addition of a and b  */
        printf("value of a and b is %d and %d is\n", a, b);
        printf("and their sum is %dn", z);
}
```

Note: You will get some surprising results. !!!

!A word of caution: Never assume that variable itself takes meaningful value, so it's your responsibility to give some valid values to your variable. [**End of caution**]!

2.9.2 Initialization of variables

As soon as the variable declaration takes place, it allocates a memory space depending upon the type of a variable. The variable can also initialize at the time of declaration. The following syntax is used for initialization of a variable at any place in the program.

```
variable_name=expression;
```

e.g ,

```
int count;
count=4;
```

Since we have declared **count** as **int** type, it takes 2 byte in memory on 16 bit machine. So the ASCII value of 4 (in 16 bit representation), which is 52, is stored in the physical address of count. The ASCII value 52 in 16 bit representation is **0000000000110100.**

Thus, pictorially, the memory content of **count** variable having value 4 is looks like:

count	00000000	00110100	⇒	**count**	4
	2003	2004			

(a) Internal memory details (b) programmer's view

The part (a) shows internal memory details for storing the value of variable **count**, while from programmer's point of view (b); the variable **count** has the value 4.

The **declaration** and **initialization (or defining)** of a variable can be done in a single step using the syntax given below:

```
data_type variable_name=expression;
```

See the following examples:

```
char letters='X';
int marks =76;
float amount = 145.13;
```

Note: it's a good programming practice to write initialized variables on a separate line followed by a comment beside the variable name.

See the following examples:

```
        int qty;                /* quantity of an item */
        floatcost = 12.10;      /* cost of an item */
        floattotal_ar1t;        /* total amount for purchase */
```

Let's look at programs 2.13 and 2.14 below and predict there outputs:

```
/* Program 2.13 */
#include<stdio.h>
int main()
{
charch1, ch2;
int var1;
printf("press any key between a-z, then press Enter key\n");
scanf("%c", &ch1);
printf("press another key between a-z, then press Enter key\n");
scanf("%c", &ch2);
printf(" The ASCII value of characters you typed are:");
printf("%d and %d\n", ch1, ch2);
var1 = ch1 * ch2;
printf("Their product is %d\n", var1);
return 0;
}
/* Program 2.14 */
#include<stdio.h>
int main()
{
        char a='H', b ='e', c='l', d='l', e='o', newline ='\n';
        printf("%c", a);
        printf("%c", b);
        printf("%c", c);
        printf("%c", d);
        printf("%c", e);
        printf("%c", newline);
        return 0;
}
```

Check your progress 2

1. Write a program to compute volume of a cone of radius (R) 10 and height (H) 15. The volume of a cone is given by the expression:

   V = (1/3) * PI * R*R*H

2. Write a program to compute simple interest; if principal amount, year for deposit and rate of interest is P, T, R respectively and simple interest I is given by.

   I = P * R * T /100[**Hint:** P, R, and T may be float values]

3. What will be the output of the following program:

```
#include<stdio.h>
void main()
{
        char a, b, c = 'd';
        b = c / 20;
        a = b * b + 12;
        printf("%c", a);
        printf("%d", a);
}
```

4. State the output of following program.

```
#include<stdio.h>
voidmain()
{
        int alpha = 077, beta = 0xabc, gamma = 123, res;
        res = alpha + beta - gamma;
        printf("%d\n", res);
        res = beta/alpha;
        printf("%d\n", res);
        res = beta % gamma;
        printf("%d\n", res);
        res = beta/ (alpha+gamma);
        printf("%d\n", res);
}
```

5. Write a C program to verify whether:

12*12 + 13*13 + 14*14 + 15* 15 + 16 * 16 = 17*17 + 18*18 + 19*19 +20 *20

2.10 SUMMARY

The concept of tokens and its various categories like; identifiers, keywords, constants, string literals, operators, and other separators are discussed. The trigraph sequences and character set of C is briefly explained. The basic data types of C are defined through the seven keywords: **int,long**, **short**, **unsigned**, **char**, **float** and **double**. (The keyword **signed** and long **double** are ANSI c extensions).

The printf() statement is defined with the help of following format conversions:

d	Decimal integers
u	Unsigned integers
o	Octal integers
x	Hex integers, lowercase
X	Hex integers, uppercase
f	Floating point numbers
e	Floating point numbers in exponential format, lowercase **e**
E	Floating point numbers in exponential format, uppercase **E**
g	Floating point numbers in the shorter of **f** or **e** format
G	Floating point numbers in the shorter of **F** or E format c single characters

UNIT 3: STORAGE CLASSES

Structure

3.0 INTRODUCTION

One of the 'C' language's strengths is its flexibility for defining the data storage. There are two aspects that can be controlled in 'C': **scope** and **lifetime**. **Scope** refers to the places in the code from where the variable can be accessed. **Lifetime** refers to the points in time at which the variable can be accessed. As we know that any variable used in the program must be declared before using it in any statement. But this is not sufficient because,to fully define a variable one needs to mention not only its '**type**' but also its '**storage**' classes. The storage class determines scope and lifetime of a variable. This unit introduces the notion and four types of storage class and also lifetime of local and global variables.

3.1 OBJECTIVES

At the end of this unit, you may be able to:

- understandthe scope and lifetime of an identifier.
- know local and global variables, and when they are used?
- storage class and its necessity to declare variable name
- know four types of storage class; **auto**, **register**, **static** and **extern**

3.2 SCOPE AND LIFTIME OF VARIABLE

Any portion of a 'C' program that is enclosed by the left brace '{' and the right brace '}' is referred as a**local block**. A 'C' function, like **main()**, contains left and right braces, and therefore anything between these two braces is contained in a local block. As we see later, **if** statement or a **switch** statement can also contain braces, so the portion of code between these two braces would be considered a local block. In addition, you may want to create your own local block without the help of a 'C' function or keyword construct. You can do this and this is absolutely legal. Variables can be declared within local blocks, but they must be declared only at the *beginning* of a local block. Variables declared in this waycan **access** only within the local block. So, we can define **scope** as a region of program text in which the variable can use or access. Duplicate variable names declared within a local block take **precedence** over variables with the same name declared outside the local block. The program 3.1 shows an example that uses local blocks:

/* Program 3.1 */

```
#include <stdio.h>
void main()
{
/* start of local block for function main() */
int x = 20;      // 1. scope of x starts here
printf("x before the inner block: %d\n", x);
{
/* start of inner local block */
intx = 10;// 2. scope of x starts here
printf("x within inner block: %d\n");
{
/* start of our own independent local block */
intx = 5;        // 3. scope of x starts here
printf("x within the independent local block:%d\n",x);
}
/* end of our own independent local block, and
end of 3. scope of x
*/
}
/* end of inner local block , and
end of 2. scope of x
*/
printf("x after the inner block: %d\n", x);
}
/* end of local block for function main(), and
end of 1. scope of x
*/
```

This example program produces the following output:

```
x before the inner block: 20
x within inner block     : 10
```

```
x within the independent local block: 5
x after the inner block: 20
```

Note:

1. It is noted that as each variable **x** was defined, it took precedence over the previously defined variable **x.** Also observe that when the inner local block is ended, the program is reentered into the scope of the original variable **x**, and its value become 10.

2. The scope of variable **x** starts from the point from which it is declared and scope ends when the local block ends.

So in general, we have two catègories of variables; **local** variable or **global** variable.

3.2.1 Local variable

These variables are declared inside some functions, like in Program3.1 (Line 10), **x** is declared in **main()** function. The **lifetime** of a local variable is the entire execution period of the function in which it is defined.Thus, the lifetime of variable **x(Line 18 of program 3.1)** is upto the execution of **Line 20**, i.e. end of independent block. Since their **scope** and **lifetime** is limited to the local block, they cannot be accessed by any other function.In general, variables declared inside a block are accessible only in that block.

Let us see another program 3.2 which computes the area and perimeter of a circle.

```
/* Program 3.2 */
/* Compute Area and Perimeter of a circle */
#include <stdio.h>
float pi = 3.14159;     /* Note here: pi is declared as global
variable*/
void main()
{
float   rad, area, peri;        /* Local variables*/
printf( "Enter the radius of circle " );
scanf("%f" , &rad);
area = pi * rad * rad;
peri = 2 * pi * rad;
printf( "Area of circle is = %f\n" , area );
printf( "Perimeter of circle is = %f\n" , peri );
}
```

The value of **pi** is defined before **main()** function, hence it can be access by any function or statement thourhgout the entire program. Thus it is a **global** variable.

3.2.2 Global variable

Global variables are declared outside all functions, like in Program 3.3(**Line 2**), **pi** is defined outside **main()** function. The **lifetime** of a **global** variable is the entire execution period of the program. It can be accessed by any function defined below the declaration, in a file.

/* Program 3.3 */

```
/* Compute Area and Perimeter of a circle */
1.      #include <stdio.h>
2.      float pi = 3.14159;  /* Global */
3.      void main()
4.      {       float   rad;    /* Local */
5.              printf( "Enter the radius " );
6.              scanf("%f" , &rad);
7.              float area = pi * rad * rad;
8.              float peri = 2 * pi * rad;
9.              printf( "Area = %f\n" , area );
10.             printf( "Peri = %f\n" , peri );
11.     }
```

The above example programs 3.1, 3.2 and 3.3 are raising a question that should variables are stored in local blocks? The use of local blocks to store the variables is unusual and therefore should be avoided, with only unusualexceptions. One of these exceptions would be for debugging purposes, when you may want to declare alocal instance of a global variable to test within your function. Another purpose to use a local block is to make your program more readable in the current context. It is suggested that you should declare your variablewhere it is used. Itmakes your program more understandable. On the other hand, well-written programstypically do not have option to declare variables in this way, and so you should avoid using local blocks.

Consider program 3.4 and predict the output with justification of your result.

/* Program 3.4 */

```
1.      #include<stdio.h>
2.      int globalVar = 1;      /* global variable */
3.      int myFunction(int);  /*This is a way to declare function
                                  prototype. */
4.      void main(void) /*              local variable: result, in main */
5.      {
6.      int result;
7.      result = myFunction(globalVar);/* call myFunction */
8.     printf("%d",result);
9.     printf("%d",globalVar);
10.
11.    }
12.
13.    int myFunction(int x)        /* Local variable x */
14.    {
15.    ++x;            /* Note: it is pre-increment of variable x */
16.    printf("%d",x);
17.    printf("%d",globalVar);
18.    ++globalVar;  /* Note: it is pre-increment of variable
       globalVar */
19.    return x;
20.    }
```

The program 3.4 has a **global** variable **globalVar** initialized with value 1. We also have a function

int myFunction(int); [See Line 3 of the above program]

The line 3 declares a function whose name is **myFunction**. Its return type is **int**, which indicates that after performing computation, this function returns **int** value. The parentheses of **myFunction** indicates that it receive an argument of **int** type.

At line 7, we call this function by writing

result = myFunction(globalVar);

This call takes **int** type argument, which is **globalVar**,and returns **int** value. The returned value is stored in a local variable **result**. The **result** is local variable because it is declared and defined in **main()** function.

The **definition** of **myFunction** is given at line 13. Here, the variable x receives the values passed by **myFunction** from line 7. So **x** receives the value of **globalVar**, which is 1.

In general, every function has:

- a prototype, also called function declaration (Line 3)
- a call to function (line 7)
- a definition of function (line 13)

We will see functions in more detail in next Block. Here, our purpose is to demonstrate how to identify scope and lifetime of local and global variable.

The **globalVar** is a global variable, its scope is throughout the program and its lifetime remains till the end of **main().** On the other hand, variable x is local to the function **myFunction**, so its scope is limited within the function and its lifetime is limited till the end of **myFunction()**. After that, it is no longer in use and hence removed from the memory. The call of**myFunction** transfers the control from line 7 to line 13. In line 15, there is a statement

```
++x;
```

This statement signifies that we have to increment the value of xby 1 before going to use **x**. Thus, the new value of **x** becomes 2. The output at line 16 is now 2. Since **globalVar** is global variable, it is still inner for the **myFunction()** block, and hencı its value at line 17 is 1. Next, we pre-increment the **globalVar** and its new value becomes 2 at line 18. And finally, we return the value of **x** from the **myFunction()**. This return the control back to line 7 of **main()** f nction. The returned value of **x** is now assigned to local variable **result** declared in **main()**. This value of result is printed in line 8 and gives 2. Similarly, value of **globalVar** is 2. [**Why?**]

Check your progress 1

1. **Predict the output of the following program.**

```
int globalVar = 1;        /* global variable */
int myFunction(int);  /* function prototypes */
void main(void)
{         int result;
          result = myFunction(globalVar); /* call myFunction */
          printf("%d",result);
}         int myFunction(int x) /* Local variables: x, globalVar */
{         int globalVar;           /* new "local" variable */
          printf("%d\n",globalVar);   /* prints ??? */
          return (x + 1);
}
```

2. **Predict the output of the following program.**int

```
myFunction(int);                /* function prototypes */
void main(void) /*              local variables: x,result in main */
{
          int result, x = 2;
          result = myFunction(x);        /* call myFunction */
          printf("%d",result);
          printf("%d",x);
}
          int myFunction(int x) /*      Local variable: x */
{
          x = x + 1;
          printf("%d\n",x);
```

```
        return x;
    }
```

3. **Predict the output of the following program.**

```
    #include <stdio.h>
    int x;                  /* x is a global variable */
    int y;                  /* y is a global variable  */
    void swap(int, int);    /* prototype for swap */
    void main(void)
{   x = 1;          /* x and y are not declared here!!! */
    y = 2;
    swap(x,y);
    printf("x = %d , y = %d \n",x,y);
}   void swap(int x , int y)
{
    int temp = x;
    x = y;
    y = temp;
}
```

Is this program really swaps the value of x and y? Justify your answer.

3.3 STORAGE CLASSES

Any variable which is used in the program must be declared before using it in any statement. Hence to accomplish this task, the type declaration statement is used. The syntax of this declaration is as:

```
data_type   var_1, var_2, . . . ;
```

To **fully define** a variable one needs to mention not only its '**type**' but also its '**storage**' class. In other words, not only all variables have a **data type**, they also have a '**storage**' class.'**Storage'** refers to the **scope** of a variable and memory allocated by compiler to store that variable. **Scope**(or **visibility**) of a variable is the boundary within which a variable can be used. Storage class defines the **scope** and **lifetime** of a variable.If we don't specify the storage class of a variable,default storage class is used by a variable.

There are basically two kinds of locations in a computer where the value of a variable may be kept i.e.: **Main Memory** and CPU **register**. It is the variable's storage class that determines which of these two locations is used for storage of variable's value.

Moreover, a variable's storage class tells us:

(a) Where the variable would be **stored**, i.e. either in **memory** or in CPU **register**.

(b) What will be the **initial value** of the variable, if the initial value is not specifically assigned?

(c) What is the **scope** of variable; i.e. in which block/function the value of the variable would be available.

(d) What is the **lifetime** of the variable; i.e. how long the variable would exists.

In general, there are four storage classes are supported by 'C':

(a) automatic storage class

(b) register storage class

(c) static storage class

(d) external storage class

3.4 AUTOMATIC STORGAE CLASS

A **variable** declared inside a function **without** any storage class specification, is by default an **automatic variable**. They are created when a function is called and are destroyed **automatically** when the function exits. Automatic variables can also be called **local** variables because they are local to a function.

Features of the variable defined to have automatic storage class are as follows:

Storage	•	Main Memory
Default initial value	•	An unpredictable value, which is often called garbage value
Scope	•	Local to the block in which the variable is defined
Lifetime	•	Till the control remains within the block where the variable is defined.

The **syntax** for declaration of **auto**matic variable is:

Syntax :auto data_type var_1, var_2, . . .;

where keyword '**auto**' refers to automatic storage class.

Example:

```
/* start of block */
{
        int year_passed;
        auto int year;
}
/*      end of block */
```

The example above defines two variables. We have not written any storage class name to variable **year_passed**. However, we explicitly mention the auto storage class to variable **year**. But in both cases, **auto** is the storage class for both the variables i.e. **year_passed** and **year**because the as default storage class of any variable is **auto**.

Consider a program 3.5 to demonstrate the use of automatic storage class.

```
/* Program 3.5   */
#include<stdio.h>
void main()
{
        auto int i=10;              /*   1.i declared inside main() block */
        {
                auto int i=20;    /*  2.i declared inside inner block */
                printf("\n\t i =%d",i);
        }                           /* end of scope for 2.      */
        printf("\n\n\t i= %d",i);       /* end of scope for 1.      */
}
```

The first **printf()** displays i =20 as scope of inner **i** is within the block whereas second **printf()**statementdisplays**i** = 10 as **i** is visible here, because its lifetime remains till end of main().

Execute program 3.6and predict the output values. What you infer from the results.

```
/* Program 3.6 */
#include<stdio.h>
void main( )
{
        auto int i=1;
        {
                auto int i=2;
        {       auto int i=3;
                printf("%d,i);
        }
                printf("%d",i);
        }
        printf("%d",i);
}
```

Consider a program segment below that illustrate the **illegal** use of **auto**.

```
#include<stdio.h>
auto int a;         /*  Illegal — auto must be within a block  */
void main ()
{
        auto int b=2;

}
```

3.5 REGISTER STORAGE CLASS

During the computation, the values of variables are transferred from main memory (RAM) to CPU. The computations are performed in processor (CPU) and the final result is sent back to RAM. This action slow down the process of execution.

On the other hand, by storing the values directly into the CPU's register, the computation speeds up. Hence it is obvious that, there should be no waste of time to getvariables from memory and sending it to back again.Therefore the **register** is used to define **local** variables and the variable will store directly in the register of CPU instead of RAM. It means that the variable has the maximum size equal to the size of register (usually one word) and can't allow to apply the '**&**' operator (as it does not have a memory location).

CPU register should be used only for the variable which require fast access, such as loop counters. It should also be noted that by defining the variable as '**register**'does not mean that the variable will always be stored in the register. It means that it may be stored in a register - depending on hardware and implementation restrictions.Becasue, there are limited numbers of CPU registers, so only few variables can be placed inside register.

The features of a variable defined as register storage class data type are as follows:

Storage	·	CPU register
Default initial value	·	garbage value
Scope	·	Local to the block in which the variable is defined
Lifetime	·	Till the control remains within the block in which the variable is defined.

The **syntax** for declaration of **register** variable is:

Syntax :register data_type var_1, var_2, . . .;

where keyword '**register**' refers to register storage class.

! A word of caution. The **data type** for **register** variable must be **int**. We cannot use **register** storage class for all types of variables. **[End of caution]**

Example:

```
{
        register int counter;
}
```

The **register**storage class hints to the compiler that the variable will be heavily used andshould be kept in the CPU's registers, if possible, so that it can be accessed quickly. There are several restrictions on the use of the **register**storage class.

(i) Because the variable **may** not be stored in memory, its address **cannot** be taken with the **unary &**operator. An effort to retrieve the address of variable will cause an error by the compiler.

(ii) The variable **must** be of a type integer to store it in the CPU's register. This implies that a single valueof a size less than or equal to the size of an integer. Some machines have registers that can hold floating-pointnumbers as well.

(iii) The number of registers are **limited**, so it may actually make the execution process slower because the register is keeping the variable and it may unavailable for other processing work.

(iv) It is **not** applicable for arrays, structures or pointers.

(v) It **cannot** not used with **static** or **external** storage class.

Consider program 3.7 to demonstrate register storage class and predict the output of the program.

```
/* Program3.7 */
#include<stdio.h>
void main()
{
        register int i=10;
        {
                register int i=20;
                printf("\n\t %d", i );
        }
        printf("\n\n\t %d", i );
}
```

The first **printf()** displays i =20 as scope of inner **i** is within the block whereas second **printf()** statement displays **i** = 10 as **i** is visible here, because its lifetime remains till end of main(). So the output of the above program is 20 and 10 respectively.

3.6 STATIC STORAGE CLASS

static is the default storage class for **global** variables. The two variables below (book and page) both have a **static** storage class as:

```
static int book;        /* explicitly declared as static
int page;               /* global variable */
{
        printf("%d\n", page);
}
```

A **static** variable tells the compiler to **persist** the variable until the end of program. Instead of creating and destroying a variable every time when it comes into and goes out ofscope,**static** variable is **initialized only once** and **retains**the value till the end of program. **Static** storage class can be used only if we want the value of a variable to **persist** between different function calls.

Features of a variable defined to have a static storage class are as follows:

Storage	·	Memory
Default initial value	·	Zero (0)
Scope	·	Local to the block in which the variable is defined
Lifetime	·	value of the variable **persists** between different function calls.

The **syntax** for declaration of **static** variable is:

Syntax : staticdata_type var_1, var_2, . . .;

Example: **static int** a;

This declaration tells to the compiler that variable a has static storage class. Hence initial value of a is 0, its location is in the main memory and its scope is "Local to the block in which the variable is defined" and lifetime exists till the end of program.

Like **auto** variables, **static** variables are also local to the block in which they are declared. The difference between them is that, the static variable does not disappear when the function is no longer active. Their values **persist**, it means that if the control comes back to the same function again the static variables have the same value as they had last time in the function.

The following example makes the concept clearer. In program 3.8, we have **abc()** function, in which **static**variable **i**is initialized with value 1. Since it is declared static, its scope and life exists till the end of program. Thus, it retains its previous value on each next function call. When first time **abc()** is called, **printf()** displays 1. Then, **i** is incremented by 1 and becomes **2**. When the control transferred to **main(),** the value of **i**will persist i.e. 2. Now in next call of **abc(),** value of **i** = 2 is displayed because compiler ignores the statement **static int i =1.** Thuswe get 1, 2, 3 as output because of three function calls of **abc()** function.

```
/* Program 3.8 */
#include<stdio.h>
main ( )
{
        abc( );
        abc( );
        abc( );
}
        abc( )
{
        static int i=1;
        printf("%d",i);
        i=i+i;
}
```

3.7 EXTERNAL STROAGE CLASS

A variable that is declared outside to any function is an **external** or **globalvariable.Global** variables remain available throughout the entire program. One important thing to remember about global variable is that their values can be changed by any function in the program.

The features of variables whose storage class has been defined as **external** are as follows:

Storage	·	Memory
Default initial value	.	Zero (0)
Scope	•	Global
Lifetime	•	As long as the program's execution doesn't come to an end.

The **syntax** for declaration of **external**(or **global**) variable is:

Syntax : externdata_type var_1, var_2, . . .;

The **extern** keyword is used before a variable name to notify the compiler that this variable is declared somewhere else. The **extern** declaration does **not** allocate storage for variables. Example for external variable declaration is:

extern int a;

External variables differ from other storage classes variables in a way that their **scopewere local**, whereas the scope of external variable is **global.** External variables are declared outside to all functions, so that they are available to all functions those want to use them.

Consider a program 3.13 where the use of global variable is illustrated.

```
/* Proram 3.13  */
#include<stdio.h>
int x=20;
voidmain( )
```

```
{
        int x=30;
        printf("%d",x);
        display ();
}
        display ()
{
        printf("\n%d",x);
}
```

The program 3.13 has a **global** variable **x** initialized with value 20. Another **local** variable **x** is initialized with value **30** in **main()** function. Here, name of variables are same, so to remove any conflict, the local variable gets **preference** over global variable. Hence first **printf()** displays value 30. Next, **display()** is called where **printf()**prints the global value of x. Now here, value of global variable **x**i.e. 20 is printed because its **scope** and **lifetime** still **remain**.

Note: The local variables get preference over the global variable when the conflict arises.

If we have multiple files and we want to define a global variable or function which will use also in other files, then ***extern*** storage class for the variable is usedin another file to provide reference for defined global variable or function. Thus, ***extern*** is used to declare a **global** variable or function in another file also.

Let us see another example where global variable from one file can be used in other file using extern keyword. Here ***extern*** keyword is being used to declare **count** in another file.

file1.c	**file2.c**
extern int count;	int count=5;
write()	main()
{	
printf("count is %d\n", count);	{
}	write();
	}

The variable count in **'file2.c'** will have a value of 5. If **file1.c** changes the value of count, the **file2.c** will see the new value.

3.8 SUMMARY

The **scope** of a variable (or identifier) is the portion of a program from where the variable is accessible. Both variables and functions have two attributes: **type** and **storage** class. The scope of a variable or function is related to its storage class. The four storage classes; auto, register, static and extern are discussed. An **auto** storage class determines the storage duration and scope of identifiers.The **static** storage class creates and initializes storage for variables when the program begins execution. Storage continues to exist until execution terminates. If an initial value is not explicitly stated, a static variable is initialized to 0. We can retain values of local variables by declaring them to be static. An **extern** storage class is used to reference identifiers in another file. By choosing storage class **register**, the programmer indicates an expectation that the program would run faster if a registercould be used for storage of variable instead of memory.

UNIT 4: INPUT AND OUTPUT FUNCTIONS

Structure

4.0 INTRODUCTION

Although our programs have implicitly shown that how to print messages, we have not formally discussed how can we use 'C' facilities to input and output data. We devote this Unit to fully explain the C input/ output facilities and show how to use them. In this Unit, we describe simple input and output formatting.'C' provides various header files that provide necessary information for the available library functions. The header file required for Input and output functions is **stdio.h**. This file can include in the program by following statement:

#include<stdio.h>

The file name stdio.h is an abbreviation for **standard input-output header** file.When the name of header file is bracketed by < and > a search is made for the header in a standard set of places.

A text stream consists of a sequence of lines; each line ends with a newline character. If thesystem doesn't operate that way, the library does whatever necessary to make it appear as if itdoes. For instance, the library might convert carriage return and linefeed to newline on inputand back again on output (K & R). A terminal keyboard and monitor can be associated only with a text stream. A keyboard is a source for a text stream; a monitor is a destination for a text stream.

4.1 OBJECTIVES

At the end of this unit, you may be able to:

- Understandhow to read a character from the keyboard through functions **getchar(), getch(), getche()**.
- Understandhow to write character on monitor screen through functions **putchar(), putch()**.
- Know about **printf()** and **scanf**() functions and their various conversion specifiers.

4.2 READING A SINGLE CHARACTER

In this section, we present some related programs for processing character data. The model of input and output supported by the standard library is very simple. Text input or output, regardless of where it originates or where it goes to, is dealt with as streams of characters. A *text stream* is a sequence of characters divided into lines; each line consists of zero or more characters followed by a newline character. It is the responsibility of the library to make each input or output stream confirm this model. The 'C' programmer using the library need not worry about how lines are represented outside the program (K & R).

(a) getchar()

One of the simplest operation is reading a **single** character from the standard input unit (usually the keyboard) can be perfomr by using the function **getchar()**.

The **usage** (or **syntax**) of getchar() is:

```
char getchar( );
```

Since it has **char** return type, its assignment takes the following form:

```
char_variable_name = getchar( );
```

char_variable _name is a valid 'C' name that has been declared as **char** type.

When this statement is executed, the computer waits until a key is pressed by the user and then assign the typed character as a value to

getchar() function, which in turn assign it to **char_variable _name.** The **getchar ( )** function accepts any character from the keyboard, this includes RETUREN and TAB. This could create problems when we use **getchar( )** interactively in a loop.

Let us look at Program 4.1 for illustration of **getchar().**

```
/* Proram 4.1  */
#include <stdio.h>        /* standard header for input/output */
main()
{
        char ch;
        printf("Enter any character\n");
        ch = getchar();   /* get a single character from the keyboard  */
        printf("You typed %c", ch);
}
```

(b) getch( )

Another function that returns the character that you typed **without** echoing it on the screen is **getch().**To make use of **getch()**, we must include the following header file:

#include<conio.h>

in your program.Here, **conio** stands for console input-output.

Note that **conio.h** and the **getch()** function described above are **not** a part of the ANSI-C standard butare available on most 'C' compilers written for DOS.

The **usage** (or **syntax**) of getch() is: $\boxed{\text{void getch();}}$

Since it has **void** return type, it does not return anything.

When above statement is executed, the computer waits until a key is pressed by the user. As soon as key is pressed, the control transfers to the next statement. The **getch ( )** function accepts any character from the keyboard and there is no need to hit Enter key.

Let us look at Program 4.2for illustration of **getch()**. The program adds two numbers and print their sum.

/* Proram 4.2 */

```
#include <stdio.h>        /* standard header for input/output */
#include<conio.h>         /* header file for getch() function  */
voidmain()
{
        int a, b, c;char ch;
        printf("Type first number\n");
        scanf("%d", &a);
        printf("Type second number\n");
        scanf("%d", &b);
        c = a + b;
        printf("The sum of %d and %d is %d\n", a, b, c);
        getch();                /* computer waits for a key to press   */
}
```

When you run the above program on your machine, its output looks like:

```
Type first number:
2
Type Secondnumber:
3
The sum of 2 and 3 is 5.
Press any key to terminate.
```

On pressing any key from the keyboard, program terminates successfully.

(c) getche()

Another function that returns the character that you typed and also **echoes** the character on the screen is **getche().** To make use of **getche()**, we must include the **#include<conio.h>** header file in your program.

The **usage** (or **syntax**) of getche() is: void getche();

Since it has **void** return type, it does not return anything.

When above statement is executed, the computer waits until a key is pressed by the user. As soon as key is pressed, the **key** is **displayed** on the screen and the control transfers to the next statement.

Let us look at Program 4.3 for illustration of **getche()**. The program 4.3 is similar to program 4.2, but note the **difference** in functioning of **getch()** and **getche().**

/* Proram 4.3 */

```
#include <stdio.h>        /* standard header for input/output */
#include<conio.h>         /* header file for getch() function */
void main()
{
        int a, b, c;char ch;
        printf("Type first number\n");
        scanf("%d", &a);
        printf("Type second number\n");
        scanf("%d", &b);
        c = a + b;
        printf("The sum of %d and %d is %d\n", a, b, c);
        getche(); /* computer waits for a key to press  and echoes
it*/
}
```

When you run the above program on your machine, its output looks like:

```
Type first number:
2
Type Secondnumber:
3
The sum of 2 and 3 is 5.
Press any key to terminate.
v
```

If you press key '**v**', then it displayed in your monitor screen as shown above and program terminates successfully.

4.3 WRITING A SINGLE CHARACTER

(a) putchar()

As we have seen that to read a single character, we need getchar() function. There is an similar function **putchar()** for writing a **single**character to the monitor screen . It takes the following form:

```
putchar(char_variable_name);
```

where**char_variable _name** is a **char**type variable containing a single character.

When the above is statement is executed, the **putchar()** function displays **char_variable_name** on monitor screen.

Let us look at Program 4.4 for illustration of **putchar().**

```
/* Proram 4.4  */
#include <stdio.h>        /* standard header for input/output */
voidmain()
{
        char ch;
        printf("Enter any character \n");
        ch = getchar();   /* get a single character from the keyboard  */
        printf("You typed: ");
        putchar(ch);
}
```

(b) putch()

The function of **putch()** is similar to **putchar()**. It prints a single character on monitor screen.It takes the following form:

```
putch(char_variable_name);
```

where **char_variable _name** is a **char** type variable, containing a single character.

When the above is statement is executed, the **putch()** function displays **char_variable_name** on monitor screen.

Program 4.5 below illustrates the use of previous discussed functions. You must include <stdio.h>to invoke these functions.

```
/* Proram 4.5  */
#include <stdio.h>        /* standard header for input/output */
voidmain()
{
        char ch1;
        printf("Enter any lowercase character (a –z)\n");
        ch1 = getchar();    /* get a single character from the keyboard  */
        printf("You typed: ");
        putchar(ch1 - 32);      /* displays uppercase character on
                                terminal screen. The ASCII codes of
                                uppercase letters are 32 less than for the
                                corresponding lower case characters
                      */
        putch('\n');
}
```

Check your progress 1

1. Execute and verify the output of the following program

```
#include <stdio.h>        /* standard header for input/output */
void main()
{
        char ch1;
        printf("Enter any lowercase character (a –z)\n");
        ch1 = getchar();   /* get a single character from the keyboard */
        putchar(ch1); putchar(ch1+2);
        putchar(ch1+4);
        putchar(ch1+6);
        putchar(ch1+8);
        putchar(ch1+10);
        putchar(ch1+12);
        putch('\n');
}
```

4.4 FORMATTED INPUT-OUTPUT

The getchar( ) and putchar( ) functions are restricted in the sense that only one character can be read-in and read-out respectively at a time and these function can not reador write float and string data types. The functions**printf( )** and **scanf( )** fall under the category of formatted console input-output functions. These functions allow us to supply the input in a fixed format and obtain the output in the specified form. The next subsequent sections describe the formatted input and output functions.

4.5 FORMATTED INPUT

An input data that is arranged in some particular format is known as formatted input. The function **scanf()** is used for formatted input from standard input device that is keyboard. The name **scanf** stands for "**formatted scan**". We have already seen this input function in a number of examples. In this section we explore several other features and options that are available for reading formatted data with **scanf()** function. The general form of **scanf ( )** statement is as follows:

int scanf("format control string", &var1, &var2,......, &varn);

The **format control string** specifies the field format in which data is to be entered from keyboard. The **format control string** can contain the following items:

(a) **White space character:** Itincludes blanks ( ), tabs (\t) and newline (\n). White spaces in the input are usually ignored. But in some situation, they are not discarded. We shall see those situation later in this unit. A white space in the format string corresponds to zero, or more white spaces that are input by the user.

(b) **Conversion (or Format) Specifiers:** The specification of **format specifier** (or **modifiers**) is shown below. The control of input conversion is much easier than for output conversions. **Any, all,**or **none** of the following format *modifiers* may be used between the % and the final **letters** of the **conversion** (or **format**) specification. It should remember that these must appear (if at all) in the sequence shown below. For readability purpose, we use **Õ**to indicate a **space** in the example output where spacing is not obvious.

Specification of **conversion** (or **format) specifier**				
%	*	maximum-field-width	length	letters **(See Table 4.1 for more details)**
	(See Table 4.2 for more details about above fields)			

The **arguments** next to **format control string** are the addresses of the variable (**&var1, &var2,......,&varn**) to which you want to write in the input. **scanf()** takes a variable number of variables, as many as there are format specifiers in the format string. The **addresses of variables are**separated by **commas**.To pass the address of a variable X, you put an ampersand (**&**) before it.

Example:scanf ("%f", &X);

NOTE: The variable type has to match its corresponding format specifier. So in the caseof this example, X has to be of type float.

[Note] that the return type of **scanf()** is **int**. What does it mean? **scanf()** reports its status by returning a value. The **value** is the number of input values successfully completed.

Consider the following Program 4.6 which reads two values. Execute the program in your machine and verify the results.

```
/* Program 4.6   */
#include<stdio.h>
voidmain()
{
        int i, j, numbers_input;
        printf("type 2 numbers: ");
        numbers_input = scanf("%d %d", &i, &j);
        /* The following block of code executes only
        when you are trying to input only one value  */
        while(numbers_input != 2) {
                printf("2 numbers needed!!! try again:");
                numbers_input = scanf("%d %d",&i,&j);
        }
printf("you enetered %d numbers and their values are ", numbers_input);
printf("\t %d and %d", i, j);
}
```

The above program needs two inputs. These two inputs are read by **scanf()**. If you are trying to input less than two values, then the condition inside while loop will become true and you are prompted again two enter two values.

Consider another statement:

scanf ("%d %d", &x, &y),

The return value corresponding to the following inputs are shown below along with their explanation.

Input no.	Data input	Return value	Explanation
1.	7 8	2	Since both 7 and 8 are integers, scanf successfully read both values and return 2
2.	7 B	1	Input 2 will change x to 7 but y will remain unchanged. So return value is 1
3.	B 7	0	For input 3; both x and y unchanged. Hence, return value is 0.

Note: if the input cannot be matched to the expected format, it is left in the buffer until it can be "consumed".

[End of Note]

Table 4.1 scanf Conversion letters and their matching types

letters	Matching Data type	Auto skip leading white space?	format	Example	Sample Matching Input
d	int	yes	decimal number	int a;long b; scanf("%d %ld", &a, &b);	-23 200
o	int	yes	octal number	unsigned int a; scanf("%o", &a);	023
x or X	int	yes	hexadecimal number	unsigned int a; scanf("%d", &a);	1A
ld	long	yes	decimal number ('l' can also be applied to any of the above to change the type from **'int'** to **'long'**)	long b;scanf ("%ld", &b);	200
u	unsigned	yes	decimal number	unsigned int a; scanf("%u", &a);	23
lu	unsigned long	yes	decimal number	unsigned long a; scanf("%lu", &a);	230
c	char	**no**	single character	char ch;scanf ("%c", &ch);	P
s	char pointer	yes	string	`char s[30];` `scanf("%29s",s)`	`hello`
a, e, f, g	float	yes	number with six digits of precision; **e** is for **scientific notation**	float a;scanf ("%f", &a);	1.2
lf, lg, le	double	yes	number with six digits of precision; **le** is for **scientific notation**	double a;scanf ("%lf", &a);	3.4
%	% (a literal)	**no**	literal	int a;scanf("%d%%", &a);	23%

Table 4.2 : scanf Format Specification Syntax

Conversion Modifier	Description	Example	Matching Input	Results
*	Assignment Supression. This *modifier* causes the corresponding input to be matched and converted, but not assigned.	int a; scanf("%*s %d", &a);/* **here %*s indicates that this value must not read** */	a: 29	a=29, return value==1
maximum field-width	This is the **maximum number of character** to read from the input. Any **remaining** input is **left** unread.	int a; scanf("%2d", &a);/* **here, maximum no. of char to read is 2** */	2345	a=23, return value =1
length modifier	A *length modifier* is used to exactly specify the type of the matching argument. Since most**types** are **promoted** to **int** or **double** a **length** modifier is rarely used. However it is used for**long** and **other types** that don't have an explicit conversion letter of their own. Note thatspecific length modifiers only make sense in combination with specific conversion letters. Usingundefined combinations causes unpredictable results. The length modifiers and their meaningsare:			
	h h specifies the argument is a **short** or **unsignedshort**.	short a; scanf("% hd", &a);	200	a=200 return value==1
	l (This is the letter *ell* and not the digit *one*.) l specifies the argument is a **long** or **unsignedlong**.	long a;, b; scanf("%ld \n%d", &a, &b);	100 276447232	100 276447232 return value==2
	L Legal **for floating point** conversions (a, A, e, E, f, F, g, and G conversion letters), specifies the matching argument is a **long double**.	long a; scanf ("%Lf", &a);	3.14	3.140000 return value==1

4.5.1 Reading Integer Numbers

Refer Table 4.1 and 4.2. The Table 4.1 indicates four different types of letters; **d**, **o**, **x** or **u**. and Table 4.2 indicates **length** modifiers **h** (for **short** or **unsignedshort**) and **l** (for **long** or **unsignedlong**). These **modifiers** can be combined with their respective **letters** to form valid conversion specifiers. So valid conversion specifiers are:

d	int
u	unsigned int
hd	short int **or**unsigned short int
l or lu	long **or**unsigned long
ld	long int

Consider the following example:

short intnum1; **/* num1 is short int */**
intnum2; **/* num2 is int */**
long int num3; **/* num3 is long int */**
unsigned long int num4; **/* num4 is unsignedlong int */**
scanf("%2hd %3d %2ld %6lu", &num1, &num2, &num3, &num4);

The data line is:

```
    23  431  24  768234
```

The *field width* of first value 23 is 2 as it consists of 2 digits. Note that in **scanf()** statement, the first conversion specifier is **%2hd**, where 2 represents the **maximum width** of number to be read and**h** represent that data input is **short int**. So *this width* matches with *field width* of the number 23. So **num1**is assigned with data value 23. Similar argument is also applied to rest three numbers, i.e. **num2**, **num3** and **num4**, which receives the value 431, 24 and 768234.

Suppose the input data is:

```
    432          2356          56          974174
```

The variable **num1** will be assigned 43 (**because of %2hd**) and **num2** will be assigned 2 (**unread part of 432**). The variable **num3** will be assigned 23 (**because of %2ld**) and **num4** will be assigned 56 (**unread part of 2356**). The rest values 56 and 974174 will be assigned to the first and second variable in the next call to **scanf().** Such types of **errors** may be **removed** if we don't use *fieldwidth* specifications. Thus, the statement:

scanf("%hd %d %ld %lu", &num1, &num2, &num3, &num4);

will read the data inputs

```
    432          2356          56          974174
```

correctly and assign 432 to **num1**, 2356 to **num2**, 56to **num3 and** 974174 to **num4**.

Execute the following program 4.7 and verify the results in your machine.

```
/* Program 4.7 */
#include<stdio.h>
```

```
main()
{
        inta, b, c, d, e, f;
        int x, y, z;
                printf("Enter 2 numbers: \n");
                scanf("%*d %d %d", &a, &b, &c);
                printf("%d %d %d\n\n", a, b);

                printf("Enter two 4-digit numbers:\n ");
                scanf("%3d %4d", &d, &e);
                printf("%d %d\n\n", d, e);

                printf("Enter 2 integers: \n");
                scanf("%d %d", &a, &d);
                printf("%d %d \n\n", a, d);

                printf("Enter a nine digit number:\n ");
                scanf("%3d %4d %3d", &x, &y, &z);
                printf("%d %d %d\n\n", x, y, z);

                printf("Enter two three digits numbers: \n");
                scanf("%d %d", &a, &d, &e);
                printf("%d %d ", d, e);
}
```

4.5.2 Reading Floating Numbers

The most common specifier for floating point numbers is %f. This specifier will acceptcharacters '0'-'9', '+', '-', and 'e'. The input can be of the form 2343.45 or 136e3.The Table 4.1 and 4.2 are referred to valid conversion specifiers for **float, double** and **longdouble**as follows:

a, f, g,	float
e	float;**e is for scientific notation**
lf, lg	double
le	double ;**le is for scientific notation**
lf	long double

A number can be skipped by using %*f, %*lf, %*Lf specifiers.

Let us see some examples for reading floating point numbers.

Satatement	**Input**	**Output**	**Explanation**
scanf("%f", &a);	21	21.0	input string "21" is converted to the float value 21.0
scanf("%e", &a);	21e-1	2.1	input string "21e-1" is converted to the fp value 2.1
scanf("%f", &a);	21 41	21.0	input string "21 41" is also converted to float value 21.0, and "**Õ** 41" is left in theBuffer. [**Note**: **Õ** represents space]

Consider the following program 4.8 and predict its output.

```
/* Program 4.8 */
#include<stdio.h>
voidmain()
{
        float a, b;
        double x, y;
        printf("Enter values for a and b");
        scanf("%f %e",&a, &b);
        printf("\n");
        printf("a = %f\n b=%f\n\n", x, y);

        printf("Enter values for x and y");
        scanf("%lf %lf",&x, &y);
        printf("\n");
        printf("x = %lf\n y=%le\n\n", x, y);
        printf("x = %lf\n y=%le\n\n", x, y);
}
```

4.5.3 Reading Strings

The format specifier for a string of characters is **%s**. This specifier accepts all non-whitespace characters. The string is **terminated** once a white space is encountered. Additionally,once the whole string is read from the input, a **NULL** character(**\0**) is appended to it to indicate end of string. But what will we do if we read aa space in our string? It is the weakness of scanf(). Therefore, we may use some another functions like, gets( ).

Satatement	Input	Output	Explanation
scanf("%s", &a);	21	21ø	input string "21" is converted to the string "21ø" (**ø** is the NULL character)
scanf("%s", &a)	21e-1	21e-1ø	input string "23e-1" is converted to the string "23e-1ø"
scanf("%s", &a);	Hello World	Helloø	input string "Hello World" is converted to the string "Helloø", and"World" is left in the buffer. [**Note**: the space is left in the buffer as well]

Note: If you input a larger string than the size of the array it isbeing written to, awful things can happen. Variables in subsequent locations in memorywill be overwritten by the string. So when using **%s** always allocate enough memory tofit all the characters of the string plus an extra location for the NULL. We will discuss this issue later when we will discuss arrays.

4.5.4 Reading characters

At last, we have reached to the discussion that indicates how to read a character using scanf. Here, the things get really confused. As we have stated previously that all specifiers ignored white spaces. So once a white space is occurred in an input, the specifier input was terminated. Moreover, any white spaces **before** the input were also ignored. Let us consider that single decimal number is required by two consecutive scanf calls. The user types in a number then hits 'enter' to signal the end of the input. That "Enter key" is actually stored in the buffer. When the next scanf is called, the first thing it sees in the buffer is the "**\n**", and since that's a white space, the function discards it and waits for the user to input a second decimal number. But this is **not** the case with **single characters**.

On the other hand, **%c** will accept **any** single character as an input, including white spaces. So if thesecond scanf had requested a single character, instead of a second decimal number, then when it sees the '**\n**' in the buffer, it would just assign '**\n**' to the corresponding variable. There are many ways to get around that problem. The easiest solution involving scanf is to put a '**\n**' in the format string of the second scanf call preceding the **%c** specifier.

So our **modified code** should be look like this:

```
        scanf("%d", &x);
        scanf("\n%c", &y);   /* note: \n before %c */
```

rather than this:

```
        scanf("%d", &x);
        scanf("%c", &y);
```

4.6 FORMATTED OUTPUT

So far we have seen the use of **printf** function to print the result and comments in the program. **printf** is also a function in the **stdio.h** file. It is used to output formatted information to the console. **printf** is alike to **scanf** in many aspects, so this section will be fairly small, simply detailing the similarities and differences between the two functions.

The general form of printf( ) function lookes like:

int printf("format control string ", var1, var2, ... varn);

The return **type** of **printf** is an **int** value. It returns the number of ASCII characters outputted to the screen, including Enter key. So you do not worry about the number of successfully outputted fields since you can simply check for output errors by reading whatis output to the console. The following example demonstrates the return value of printf statement.

```
/* Program 4.9  */
#include<stdio.h>
voidmain()
{
        int i;
        i=printf("Hello World");
        printf("Number of characters output by printf is: %d", i );
}
```

The output of the above program is;

```
Number of characters output by printf is: 11
```

The variables which are printed on screen are represented by **var1**, **var2**, ….and**varn**. There is no need to pass the addresses of variables. In place of list of variable, it can be written as constants, single variable o1 an array names. Even more complex expressions, function reference may also be included.

The **format control string** specifies the field format in which data is to be displayed on monitor screen. The **format control string** can contain three types of items:

a. Characters that are simply printed as they are.

b. Escape sequences that begin with a \sign; like \n, \t, \b etc.

c. Conversion (or format) specification that begins with a %sign.

The specification of **format specifier** (or **modifiers**) is shown below. To control the appearance of the converted arguments, **any orall**, or **none** of the following format *modifiers* may be used between the % and the final **letters** of the **conversion** (or **format**) specification. It is also remembered that these must appear (if at all) in the sequence shown below. For readability purpose, we use **Õ** to indicate a **space** in the example output where spacing is not obvious.

Specification of **conversion** (or **format) specifier**						
%	flags	Minimum-field-width	.	precision	length	letters
		(See Table 4.4 for more details about				**(See Table 4.3 above fields) for more details)**

The Table 4.3 and 4.4 describes the various format control with some small examples.

Table 4.3: printf Conversion Letters and Matching Types			
Letter	Type of Matching Argument	Example	Output
%	*none*	printf("%%");	%
d	int	printf("%d", 17);	17
u	unsigned int (*Converts to decimal*)	printf("%u", 17u);	17
o	unsigned int (*Converts to octal*)	printf("%o", 17);	21
x	unsigned int (*Converts to lower-case hex*)	printf("%x", 26);	1a
X	unsigned int (*Converts to upper-case hex*)	printf("%X", 26);	1A
f, F	double	printf("%f", 3.14);	3.140000
e, E	double	printf("%e", 31.4);	3.140000e+01
g, G	double	printf("%g, %g", 3.14, 0.0000314);	3.14, 3.14e-05
a, A	double	printf("%a", 31.0);	0x1.fp+0
c	int	printf("%c", 65);	A
s	*string*	printf("%s", "Hello");	Hello
p	void*	int a = 1; printf("%p", &a);	0064FE00
n	int*	int a; printf("ABC%n", &a);	ABC (*a*==3)

Table 4.4 printf Conversion Specification Formatting			
[Note: means a space]			
Format Control	**Description**	**Example**	**Output**
flags	The flag characters may appear in any order and have the following meanings:		
	- left-justify within the field	printf("|%3d|%-3d|", 12, 12);	| 12|12 |
	+ Forces positive numbers to include a leading plus sign.	printf("%+d", 17);	+17
	space Forces positive number to include a leading space.	printf("|%**Õd**|", 12);	| 12|
	# This flag forces the output to be in some *alternate form*.	printf("%#X", 26);	0X1A
	0 Pad with zeros rather than spaces	printf("|%04d|", 12);	|0012|
minimum field-width	*field width* represents the minimum number of characters in the resulting string. If the converted value has fewer characters, then the resulting string is *padded* with spaces (or zeros) on the left (or right) by default (or if the appropriate flag is used.)	printf("|%5s|", "ABC");	| ABC|

.precision	A period by itself implies a precision of zero. The meaning of a precision depends on the type of conversion done. Only the conversions listed below are defined:		
	When used with floating-point conversion letters (a, A, e, E, f, F, g, and G) theprecision specifies how many digits will appear to the right of the decimal point. The default precision is six.	printf("|%5.2f|", 3.147);	| 3.15|
	When used with integer conversion letters (d, i, o, u, x, and X) the precision specifies the minimum number of digits to appear. Leading zeros are added as needed.	printf("|%6.4d|", 17);	| 0017|
	When used with string conversions (letter "s") the precision specifies the maximum number of bytes written. If the string is too long it will be truncated.	printf("|%-5.3s|", "ABCD");	|ABC |
length	A *length modifier* is used to exactly specify the type of the matching argument. The length modifiers and their meanings are:		
	h specifies the argument is a **short** or **unsignedshort**.	printf("%hd", 300);	300
	l (This is the letter *ell* and not the digit *one*.) l specifies the argument is a **long** or **unsigned long**.	long a = 300, b = (long) 1.0E+14; printf("%ld\n%d", a, b);	300 276447232
	L Legal for floating point conversions (a, A, e, E, f, F, g, and G conversion letters), specifies the matching argument is a **long double**.	printf("%Lf", 3.14L);	3.140000

5. What is the output of the following program:

```
main( )
{
        int Rupees = 2;
        int paisa = 3;  /* $2.03 */
        printf("$%d.%d ", Rupees, paisa);
        printf("$%d.%2d ", Rupees, paisa);
        printf("$%d.%02d ", Rupees, paisa);
}
```

4.7 SUMMARY

There are several functions that have more or less become standard for input and output operations in 'C'. These functions together are referred by standard input-output library. We have discussed some common functions that can be used for reading and writing input data. The unit discusses various functions like getchar(), getch(), getche(), putchar() for reading/writing a single character. We have also seen formatted I/O functions like printf and scanf with various conversion specifiers. Sufficient number of examples are also provided so that you are more familiar with them.

U.P.Rajarshi Tandon Open University, Allahabad

UGCS-102
Problem Solving Through "C"

Block 3

Operator and Control Structures

Block-3 INTRODUCTION

This block introduces various types of operators with their priorities, precedence and associativity. The 'C' programming language contains a rich set of operators. Operators are the symbols which have a predefined meaning (or operation) associated with them. These operations are for the mathematical or logical manipulations. A program uses operators to manipulate items or variables.

There are many situations in which the order of execution of these statements need to change based on certain specific condition or repeat a group of statements until specific conditions are fulfilled. This require a sort of decision making to check whether a particular condition is satisfied or not and then instruct the computer to execute particular instructions accordingly. The various decision structures like **if, if....else**, **nested if....else**, **switch** and **goto** statements in 'C' are describe in Unit 2.

Most algorithms require a control structure that will allow us to repeat certain lines of code until a condition is reached. We call these **repeating** structures as**loops**and discussed in Unit 3. 'C' provides three loop structures to control the repeated execution of one or more statements. The counter controlled; **for** loop is a counting loop that may also involve other conditional testing, used extensively with arrays and uses pretesting of the loop control variable(s). The **while** or **do-while** loop is used for event controlled repetition. A **while** loop is used when it is possible that the loop may never execute. It is a conditional loop that often doesn't involve counting, it uses pretesting of the loop control variable(s). A **do-while** loop is used when the loop must execute **at least** one time. It uses post-testing of the loop control variable(s) and generally used with interactive input. C allows us to use *jump statements*. These statements, like break and continue, are also flow control statements that cause the interruption of the execution flow and jumps to a different statement that is not lie in the successive sequence path of program statements.

In Unit 4, we will see one-dimension and multidimensional array, which are used to hold a group of variables of the same type. Each element of the array is identified and accessed by its subscript or position in the array. Array subscript begins at 0 for the first element. The array concept makes possible random access to any element. A string is a group of characters usually letters of the alphabet. 'C' uses a string of data in some way, either to compare it with another string, output it, copy it to another string, or whatever, the functions are set up to do what they are called to do until a null, which is a zero, is detected.We have also studied some standard predefined functions which are available for use. These are mostly input/output functions, character and string manipulation functions.

UNIT-1 Operators and Expressions

Structure

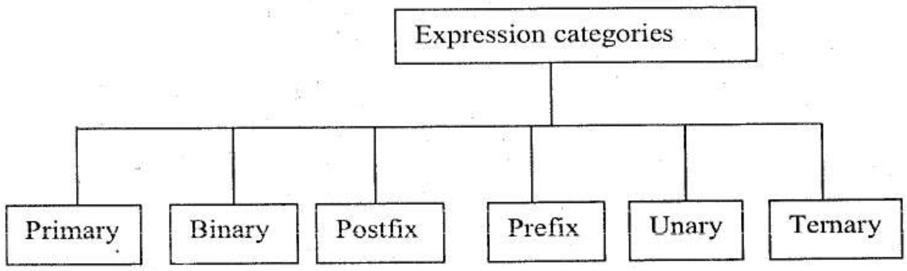
1.0 INTRODUCTION

This unit introduces various types of operators with their priorities, precedenceand associativity. The 'C' programming language contains a rich set of operators. Operators are the symbols which have a predefined meaning (or operation) associated with them. These operations are for the mathematical or logical manipulations. A program uses operators to manipulate items or variables. The data items or variables on which operators operate are known as **operands**. The operators of 'C' language can classify into the following categories:

- Arithmetic operators
- Relational operators
- Logical operators
- Assignment operators
- Increment and decrement operators
- Conditional operators
- Bitwise operators
- Special operators

An **expression** is a sequence of operands and operators that reduces to a single value. Expressions can be simple or complex. An **operator** is a syntactical token that requires an action be taken. An

operand is an object on which an operation is performed. The expressions are categorized into the following categories:



The primary expressions are consisting of variable names, literal constants and parentheses expressions. Some of the examples of primary expressions are:

- Names: height, volue, price, INT_MAX, SIZE
- Literal constants: 5, 345.34, 'C', "Hello"
- Parentheses expressions: (4 *3 / 6), (b = 24 + c % 2)

We will discuss more about rest of the expressions later in the next subsequent sections while discussing operators, as each type of expression particularly work by a specific types of operators categories.

1.1 OBJECTIVES

After working though this Unit, you should be able to:

- Write and evaluate complex 'C' language expressions, built with the arithmetic, logical and other complex operator categories.
- Know the concept of precedence and associative properties among various operators, and how they are evaluated.
- How these properties will help you to decide sequence in which the various statements of a C program are evaluated.

1.2 ARITHMETIC OPERATORS

Arithmetic operators are the most commonly used operator. They are used to perform arithmetic operations and operate on any built-in data types allowed in C. These operators are:

Table 1.1: Arithmetic operators

Operator	**Meaning**	**Examples**
+	Addition or unary plus	5 + 7
-	Subtraction or unary minus	Count – 1
*	Multiplication	num1 * num2
/	Division (divisor must by non zero)	num / den
%	Modulo division (gives remainder. This operator is valid only for integer division)	Count % 3

The operator- can be used as unary minus (-) operator. The unary minus negates the sign of its operand. For example; -(-1) = 1. In C, **all** numeric constants are positive. Therefore, in C, a negative number is actually a positive constant preceded by aunary minus, for example: -3.

Note:'C' does not any operator for ***exponentiation***.

All of these operators are binary operators. Here the term **binary** means we need two operands. The two operands and an operator form a **binary (or arithmetic)expression**. For example, consider the expression:

(a - b) * (a + b) /4

We have already discussed in previous blocks that 'C' is a weak typed language. The **weak** typing of 'C' supports the **mixing** of operands of differing types. Operands may undergo type conversion before an expression takes on its final value. The general **rule** is that the final result will be in the highest precision possible given the data types of the operands.

When an arithmetic expression is assigned to a variable, there are two operators used i.e. an arithmetic operator and an assignment operator. Let us consider following expression:

A = B + C;

First **B**and **C**are added, the result will be in the same precision as the highest precision of **B**and **C**. This result will be then **cast** (or **converted**) into the precision of **A**and the assignment will occur. The **expression** becomes a **statement** because it is terminated with a semicolon.

Note: The **left** operand in an **assignment expression** must be a **single** variable.

An arithmetic statement is of following types:

1.2.1 Integer arithmetic expression

This type of expression consists of either integer constant or integer variable; i.e. both operands are integer type. The result of operation always yields integer value. Consider the following example 1:

Example 1: Integer arithmetic expressions		
Assume a = 22, b = 15		
statement	**Result**	**Explanation**
a + b	= 37	As usual, no need to explain
a – b	= 7	"
a * b	=330	"
a / b	=1	Decimal part truncated
a % b	= 7	Remainder of division

In integer divisor, following points are noted:

- If both operands have same sign, then result is truncated towards zero.i.e. 2/3 = 0 or -2/-3= 0.
- If **one** of them is **negative,** the truncation is implementation dependent. i.e. -7/8 may be zero (0) or -1.

Similarly, in modulo division, the sign of the result is always the sign of first operand (i.e. dividend). For example:

```
-16 % 3   = -1
- 16 %.-3 = -1
 16 % 3   =  1
```

Note: Both operands of the modulo operator (%) must be integer types.

Question 1. Execute the following Program 1.1 and verify its result.

```
/ * program 1.1 */
#include<stdioh>
voidmain()
{
        int minutes, seconds;
        printf("enter seconds\n");
        scanf("%d", & seconds);
        minutes = seconds/60;
        seconds = seconds%60;
        printf("minutes = %d seconds = %d", minuts, seconds);
}
```

1.2.2 Floating point Arithmetic Expression

This type of expression consists of **floating point** (or **real**) operands(either **real constant** or **real variables**). The real operands may be either in decimal or exponential notation. The result of operation is always an approximation of the calculation performed. Consider the following example 2:

Example 2: floating Point (or real) arithmetic expressions		
Assume a = 17.67, b = 5.1		
statement	**Result**	**Explanation**
a + b	= 22.77	As usual, no need to explain
a – b	= 12.57	"
a * b	= 90.116997	Precision upto 6 digits
a / b	= 3.464706	Precision upto 6 digits

Note: We cannot use % operator with real numbers.

Question 2: Execute the following Program 1.2 and verify its result.

```
/ * program 1.2 */
#include<stdioh>
voidmain()
{
        float cent, faren;
        printf("enter temperature in centigrade\n");
        scanf("%f", &cent);
        faren = 1.8*cent +32.0;
        printf("temperature in centigrade is %f", cent);
        printf("temperature in Farenhite is %f",faren);
}
```

1.2.3 Mixed Mode arithmetic Expression

When one operand is real and another is integer type then, such an expression is called **mixed mode** arithmetic expression. If any one of the operand is real, then real operation is performed, and the result is always a real number. Consider the following example 3:

Example 3: Mixed Mode arithmetic expressions		
Assume a = 17.67 (a real number), b = 5 (a integer number) c (a integer number) [NOTE]		
statement	**Result**	**Explanation**
c=a + b	= 22	See [**Note**] below
c=a – b	= 12	
c=a * b	= 88	
c= a / b	= 3	

Note for Example 3: First **respective operation** on **a** and**b**are performed, the result will be in the same precision as the highest precision of **a**and **b**. This result will be then **cast** (or **converted**) into the precision of **c**and the assignment will occur.

Let us consider another example 4 where variable **c** is of **real** type.

Example 4: Mixed Mode arithmetic expressions		
Assume a = 17.67 (a real number), b = 5 (a integer number) c (a real number) [NOTE]		
statement	**Result**	**Explanation**
c=a + b	= 22.670000	The explanation for example 4 is same as given in example 3.
c=a – b	= 12.670000	
c=a * b	= 88.350000	
c= a / b	= 3.534000	

Consider the program 1.3 and execute the program in your machine. The program takes three numbers as input, calculate their sum and average.

```
/ * Program 1.3 */
#include<stdioh>
voidmain()
{
        int num1, num2, num3, sum;
        float avg;
        printf("enter three integer numbers \n");
        scanf("%d%d%d", &num1, &num2, &num3);
        sum = num1 + num2 + num3;
        avg = sum/3.0;
        printf("entered numbers are %d %d%d\n", num1, num2, num3);
        printf("Their sum and average is %d and %f\n", sum, avg);
}
```

The output of the above program is:

```
Enter three integer numbers
3 5 7
Entered numbers are 3 5 7
Their sum and average is 15 and 5.000000
```

In 'C', the arithmetic operators have the priority as shown below:

First priority	* / %	Arithmetic operators
Second priority	+ -	
Third priority	=	Assignment operator

Operators with within the **same priority** are evaluated from **left** to **right**.

1.3 RELATIONAL OPERATORS

This category of operators is used to **compare**arithmetic expression, functions, variables and constants. For example, we often compare weight of two persons, or price of two items. This comparison can be done with the help of **relationaloperators**. The **relational** operators are also **binary** operators as they require two operands, i.e. **operand 1** and **operand 2**.

Here, **operand 1** and **operand 2** may be either:

- Arithmetic expression
- Return value of Functions
- Variables
- Constants

and, **operator** may be any one of the operator as mentioned in Table1.2. When arithmetic expressions are used on either side of a relational operator, the arithmetic expressions will be evaluated first and then the result will compare. Thus, arithmetic operators have higher precedence than relational operators.

For example; an expression

a < b

that contains a relational operator is referred as **relationalexpression**. The result of relational expression is always either **1 (TRUE)** or **0(FALSE)**.

'C' has following six relational operators as shown in Table 1.2.

The relational expressions are used in decision making statements. These will later discuss in next units.

Table 1.2: Relational operators

Operator	Meaning	Examples	Result of relational expression
<	Is less than	3 < 2	TRUE
<=	Is less than or equal to	20 <= 20	TRUE
>	Is greater than	5 > 7	FALSE
>=	Is greater than or equal to	7 >= 9	FALSE
==	Equal to	4 == 4	TRUE
!=	Not equal to	2 != 3	TRUE

Note: In 'C', **0** represents **FALSE** and **1** represents **1(TRUE)**.

What does the following program print?

```
#include<stdioh>
voidmain()
{
        printf("%d\n", 5 > 3);
        printf("%d\n", 3 < 5);
        printf( "%d\n", 4 + 3 >= 2 +1 );
}
```

1.4 LOGICAL OPERATORS

'C' has following three logical operators as shown in Table 1.3. The logical operators && and | | are used when we need to test more than one condition and make some decision base on the obtained result.

Table 1.3: Logical operators

Operator	Meaning	Logical	Description expression
&&	Logical AND	Exp1 && Exp 2	The logical expression is true only if both (Exp1 and Exp 2) Expressions are true.
| |	Logical OR	Exp1 | | Exp 2	The logical expression is true only if either (Exp1 or Exp 2) Expressions are true.
!	Logical NOT	! Exp1	It is just the negate of current value of Exp1

An expression that consists of two or more relational expressions is termed as **logicalexpression** or a **compoundrelationalexpression**. Consider an logical expression given below:

x < y && i == 5

Like relational expression, logical expression also gives the **result** in terms of **1**(**TRUE**) or **0**(**FALSE**).

The **logical** operators (except *logical***NOT** (**!**)) are also **binary** operators because they require two operands. When relational expressions are used on either side of above saidlogical operators, the relational expressions will be evaluated first, which in turn evaluates arithmetic expression and then the result will be computed. Thus, arithmetic operators have higher precedence than relational operators, which have higher precedence than logical operators (except *logical***NOT**).[See Table 1.4].

Table 1.4: Result of Logical AND and OR Expressions

Operand 1	Operand 2	Value of logical AND	Value of logical OR
		Operand 1 && Operand 2	Operand 1 || Operand 2
Non zero	Non zero	1	0
Non zero	0	0	1
0	Non zero	0	1
0	0	0	0

The *logical***NOT** operator takes only **one** operand, so it is come under the category of **unaryoperators.** The expressions formed by unary operators are termed as **unaryexpressions**. This unary expression is also called **prefix** expression, as operator always lies **left** to the expression.**[See table 1.5].**

Table 1.5: Result of Logical NOT expression

Value of logical NOT	
Operand 1	**! operand 1**
Non zero	0
0	1

Consider the following example 5 that involves simple logical statements to complex logical statements. It is an exercise for you to justify the evaluated part and answer how the respective results are obtained.

While evaluating complex logical expressions joined by !, && and | |, there is no need to evaluate the entire expression logical expression if its value is found from its sub expressions. Consider an example:

0.003 > 1.23 && 30 > 12

Here first sub expression (0.003 > 1.23) is evaluated, which is FALSE. So there is no need to evaluate sub expression (30 > 12), because the value of whole expression will be FALSE due to the presence of logical AND operator.

Example 5: Logicalexpressions
Assume a = 4, b = 5, c = 7, d = 12, e = 56, f = 34

Logical expression	Operand 1	Operand 2	evaluation	Output result
((a*b) < (d+e)) && (c > d)	relational expression	relational expression	(20 < 68) && (7 > 12)	(0) FALSE
((e/a) >= 3) && (a)	arithmetic expression	variable	(14) && (4)	(1) TRUE
-23 || ((c+d)<= (d+f))	constant	relational expression	-23 || (19 <= 46)	(1) TRUE

Let us consider another example

0.003 > 1.23 | | 30 > 12

Here, first sub-expression is evaluated which results FALSE. Next second sub-expression is evaluated due to the presence of | | operator as the result of whole expression cannot be decided without evaluation of both sub expressions. The evaluation of both sub expressions gives the result TRUE.

What does the following program print?

```
#include<stdioh>
voidmain()
    {
        printf("%d\n",7>3);
        printf("%d\n",8 <5);
        printf( "%d %d\n", (5>3) && (3<5), (5>3 ) || (3<5));
        printf( "%d %d\n", (1> 0) * (1<0),  1>0  * 1<0);
    }
```

1.5 ASSIGNMENT OPERATORS

Assignment operator (=) is used to assign the value of an expression to a variable . An assignment operator is also a binary operator that requires two operands. The left hand operand is called as **lvalue.** Similarly, right hand operand is called as **rvalue**. An lvalue is an expression to which a value can be assigned. The **lvalue** expression is located on the *left side* of an assignment statement, whereas an rvalue is located on the *right side* of an assignmentstatement. Each assignment statement **must** have an lvalue and an rvalue. The **lvalue**expression mustreference a **storable** variable in memory. It **cannot** be a constant.

The 'C' language offers number of compound operators, including assignment operator that combine other operations, as summarized in Table 1.6. These **compoundoperators** are also known as **shorthand** assignment operators. They have following form:

varop= exp;

Where var is a**lvalue** variable, **exp** is an expression and **op** is a binary arithmetic operator. The operator **op=** is known as **shorthand** (or **compound**) assignment operator.

The above statement is similar to **var = var op exp**

Thus, the statement **x = x +1** is similar to **x +=1.**

Table 1.6: Shorthand (or compound) assignment operators

Operator	Description	Example	Remark
=	Assignment	Num = 6	Assign 6 to num
+=	Sum and assign	Num += 6	Adds 6 to number
-=	Subtract and assign	Num -= 16	subtract 16 to number
*=	Multiply and assign	Num *= 5	Same as Num = Num*5;
/=	Divide and assign	Num /= 13	Same as Num = Num/13
&=	Bitwise AND and assign	Num1 &= Num2	**ands** the bits of Num1 with Num2
|=	Bitwise OR and assign	Num1 |= Num2	**ors** the bits of Num1 with Num2

Question 3: See the program below and check how the complex assignment can be done.

```
int main()
{
 int a, b, c;  /* Integer variables for examples */
  a = 12;
  b = 3;
  c = a + b;              /* simple addition           */
  c = a - b;              /* simple subtraction         */
  c = a * b;              /* simple multiplication      */
  c = a / b;              /* simple division           */
  c = a % b;              /* simple modulo (remainder)    */
  c = 12*a + b/2 - a*b*2/(a*c + b*2);
  c = c/4+13*(a + b)/3 - a*b + 2*a*a;
  a = a + 1;              /* incrementing a variable     */
  b = b * 5;
  a = b = c = 20;         /* multiple assignment        */
  a = b = c = a + b * c/ 3;
  a = (b = (c = 20));            /* Identical to multiple assignment        */
 return 0;
}
```

1.6 INCREMENT AND DECREMENT OPERATORS

'C' has **unary** operators for both **increment** and **decrement**. The Table 1.7 show the increment (++) and decrement operator (—).

Table 1.7: Increment and Decrement operators

Operator	Description	Example	Expression	Equivalent to
++	increment	x++	Postfix expression	x = x +1
		++x	Prefix expression	x = x +1
—	decrement	x-	Postfix expression	x = x -1
		-- x	Prefix expression	x = x -1

The operator ++ adds 1 to the operand while — subtract 1 from the operand.

The expressions formed by **post-increment** of ++ and - - are called **postfix** expression while expression formed by **pre-increment** of ++ and - - are called **prefix** expression. Since both expressions consider only **one** operand, that's why they come under the category of **unary** expression.

Effect of post-operator (**x++** or **x- -**) and pre-operator (**++x** or **- - x**):

1. If they are **independently** used, their behavior is very simple

   For example: x++ means x = x +1 , x- - means x = x -1, ++x means x = x +1 and - -x means x = x -1.

2. If they are used as **rvalue**, i.e. right hand side of expression, their behavior is entirely different. For example; consider the following two cases:

(a)

```
x = 4;
y = ++x;
```

In this case, **x** is first incremented by 1. This incremented value of **x (i.e. 5)** is then assigned to **y**. So the value of **x** and **y** is 5.

(b)

```
x = 4;
y = x++;
```

In this case, **x** is first assigned to **y**. After that, its value is incremented by 1. So the value of xis 5 and that of**y** is 4.

Question 4: Execute the program 1.4 and verify the results in your machine.

```
/ * Program 1.4 */
#include<stdioh>
voidmain()
{
    int x=4, y=6, z;
    printf("value of x      : %2d\n", x);
```

```
        printf("value of x++    : %2d\n", x++);
        printf("new value of x: %2d\n", x);
        printf("\n\n");
        printf("value of y      : %2d\n", y);
        printf("value of ++y  : %2d\n", ++y);
        printf("new value of y: %2d\n", y);
        x = ++x + y- -;
        printf("new value of x          : %2d\n", x);
        printf("new value of y          : %2d\n", y);
        y = ++y + x- -;
        printf("new value of x          : %2d\n", x);
        printf("new value of y          : %2d\n", y);
        z = x- - - ++y + - - y + x;
        printf("value of z       : %2d\n", z);
        printf("new value of y          : %2d\n", y);
        printf("new value of x          : %2d\n", x);

}
```

1.7 CONDITIONAL OPERATOR

The **conditional** operator (or **ternary** operator) can be used to return a value. It has the following syntax:

expr1 ?expr2 : expr3

where exp1, exp2 and exp3 are expressions.

The expression ***expr1***is evaluated first. If it is non-zero (true), then the expression ***expr2***is evaluated, and that is the value of the conditional expression. Otherwise ***expr3***is evaluated, and that is the value. Only one of ***expr2***and ***expr3***is evaluated. The expression formed by ternary operator **?:** is known as **ternaryexpression.**

Consider the following example:

```
int a=6, b =7, x;
x= (a > b) ?a  : b;
```

In this case, ***exp1*** (a > b) is evaluated and becomes FALSE, so **x** is assigned value **b i.e. x= 7**.

The ternary operator is a substitute of **if…else**. The brackets round the condition are not strictly necessary, but they do aidreadability.

Consider the program 1.5 and execute it on your machine to verify the results.

```
/ * Program 1.5 */
#include<stdioh>
voidmain()
{
    int age;
            printf("Enter your age\n");
            scanf("%d", age);
            printf("your age is   : %d \n", age);
            (age> 18) ? printf("You are major");: printf("You are minor");
}
```

The output of the above program is:

```
Enter your age
43
Your age is   : 43
You are major
```

1.8 BITWISE OPERATORS

'C' has a rich set of bitwise operators to interact with the hardware, i.e. machine level. Using these operators we can access the individual bits of a byte. These operators allow us to manipulate the individual bits of a byte. The manipulation involves testing of bits, or shifting them right or left. The bitwise operators may not be applied on **float** or **double**. The bitwise operators available in 'C'are shown in Table 1.8:

Table 1.8: Bitwise operators

Operator	operation
&	**BitwiseAND**; compares two bits and generates a 1 result if both bits are 1; otherwise it returns 0.
|	**BitwiseOR**; compares two bits and generates a 1 result if either or both bits are 1; otherwise it returns 0.
^	**BitwiseexclusiveOR**; compares two bits and generates a 1 result if the bits are complementary; otherwise it returns 0.
~	**Bitwisecomplement**; inverts each bit. ~ is also known as 1's complement.
>>	**Bitwiseshiftright**; moves the bits to the right, discards the far right bit and if unsigned assigns 0 to the left most bit, otherwise sign extends.
<<	**Bitwiseshiftleft**; moves the bits to the left, it discards the far left bit and assigns 0 to the right most bit.

Execute the program 1.6 and verify the result in your machine.

```
/ * Program 1.6 */
#include<stdioh>
voidmain()
{       intx=12, y=34;
        printf("Bitwise AND of x and y is    :", x =x & y);
        printf("Content of x is now          :", x);
        printf("Bitwise OR of x and y is     : ", x = x | y);
        printf("Content of x is now            :", x);
        printf("Bitwise Ex-OR of x and y is  :", x=x ^ y);
        printf("Content of x is now            :", x);
        printf("Bitwise Complement of x is  :", ~x);
        printf("Content of x is now            :", x);
        printf("Bitwise Complement of y is  : ",    ~y);
        printf("Content of y is now            :", x);
        printf("Bitwise left shift of x <<2 is  : ", x =x << 2);
        printf("Content of x is now            :", x);
        printf("Bitwise right shift of y>>2 is : ", y= y >> 2);
        printf("Content of y is now            :",y);
}
```

Thc output of the above program is:

```
Bitwise AND of x and y is      :   0
Content of x is now            :   0
Bitwise OR of x and y is       :   34
Content of x is now            :   34
Bitwise Ex-OR of x and y is    :   0
Content of x is now            :   0
Bitwise Complement of  x is    :   255
Content of x is now            :   255
Bitwise Complement of y is     :   219
Content of y is now            :   219
Bitwise left shift of x <<2 is :   152
Content of x is now            :   255
Bitwise right shift of y>>2 is :   54
Content of y is now            :   255
```

1.9 SPECIAL OPERATORS

'C' provides some special operators which are as follows:

(a) Comma (,)operator
(b) sizeof (**sizeof()**)operator
(c) address (&) operator
(d) dereferencing (or value at address, *) operator
(e) dot (.) operator
(f) member selection (**->**) operator

We will discuss first two operators in this section, rest operators will discuss in subsequent units of pointers and structures.

(a) Comma operator

The comma operator can be used to link the related expression together. A comma separateslist of expression, these are evaluated **left-to-right** and the value of **right-most expression** is the value of the combined expression.

For example:

```
value = (x=10,y=5, x+y);
t=x,x=y,y=t;   //exchanging value
```

(b) Sizeof operator

It is a **unary** operator and also a **compiler time operator**. The **sizeof**() is a **special operator** that returns number of bytes taken by a **characterconstant**, **variable** or **data** type.

The syntax of **sizeof** operator is

sizeof(operand)

The operand may be a **character constant**, **variable** or **data** type or any **userdefineddata** type.

For example: consider the following program 1.7:

```
/* Program 1.7 */
#include<stdio.h>
        int main()
{       int a=2;
        float b =3.4;
        char ch ='s';
        printf("sizeof(a) == %d \n", sizeof(a)); /* operand is variable */
        printf("sizeof(char) == %d \n", sizeof(char)); /* operand is data
                                                             type */
        printf("sizeof('s') == %d \n", sizeof('s'));/* operand is constant */
        printf("sizeof(float) == %d \n", sizeof(float));/* operand is data
                                                             type */
        return 0;
}
```

The output of the following program is:

```
sizeof(a) == 2
sizeof(char) == 1
sizeof('s') == 1
sizeof(float) == 4
```

Check your progress 1

1. Verify by executing a 'C' program that for **int** variables **a** and**b**, **a % b** equals **a – (a / b) * b**, regardless of whether **a** or **b**are negative or positive.

2. Find the value that is assigned to the variables x, y, and z when the following proram is executed.

```
#include<stdio.h>
void main()
{       int x, y, z;
        x = 3 +4 -5 +6 – (7 – 8);
        y = 2* 34 + 4 * (6 – 7);
        z = 3 * 4 + 5 / 15 % 14;
}
        Add printf at appropriate places.
```

3. Give the output of the following program:

```
#include<stdio.h>
void main()
{       int x = 3, y = 5, z = 7, w;
        w = x % y + y % x – z % x – x % z;
        printf("%d\n", w);
        w = x / z + y / z + ( x + y ) / z;
        printf("%d\n", w);
}
```

4. What does the following program print.

```
#include<stdio.h>
void main()
{
        printf("%d\n", -1 + 2 – 12 * -13 / -4);
        printf("%d\n", -1 % - 2 + 12 %  -13 %  -4);
}
```

1.10 OPERATOR PRECEDENCE AND ASSOCIATIVITY

Precedence of operators is a very important concept that you should study. In this section we will describe the precedence and associativity of operators. When you have mixed arithmetic expressions, the multiplicationand division operations are completed before the addition and subtraction operations when they are all atthe same logical level. Therefore when evaluating **a * b + c / d**, themultiplication and division are donefirst, then the addition is performed. However in the expression **a * (b + c / d)**, the addition follows thedivision, but precedes the multiplication because the operations are at two different logical levels asdefined by the parentheses. While a parenthesis allows us to change the order of priority, we can use them to improve understandability of the program. It is always a good practice to use pair of parentheses to make sure that the priority assumed is the one we desire.

Consider a program 1.8 which shows the effect of precedence on an expression.

```
/* Program 1.8 */
#include<stdio.h>
int main()
{       int a = 10;
        Int b = 20;
        Int c = 30;
                printf("a * b + c  is  : %d\n", a * b + c);
                printf("a * (b + c)  is  : %d\n", a * (b + c));
        return 0;
```

}

The output of the following program is:

```
a * b + c is   :     230
a * (b + c ) is :    500
```

The program clearly shows that expression within the parentheses is first evaluated and then next expression is evaluated.

Precedence is used to determine the order in which different operators in a complex expression are evaluated. The precedence of C operators dictates the order of calculation within an expression. The precedence of the operators introduced here is summarized in the Table 1.10. The highest precedence operators are given first. **Associativity** is used to determine the order in which operators with the **same** precedence are evaluated in a complex expression. Associativity is applied when we have more than one operator of the same precedence level in an expression.

Now, let us see an example on associativity. Example 6 shows the right to left associativity.

Example 6:

Consider the statements:

```
int a = b = c = d = 10;
a += b *= c -= 5;
printf("a=%d b=%d c=%d d=%d\n", a, b, c, d);
```

Here the operators +=, *= , -= belong to same precedence level. So, evaluation of statement begins from righ to left. The value of c becomes c = c -5 = 10 -5 = 5. This value 5 is assigned to its lvalue i.e. b *= 5. The value of b becomes b = b *5 = 10*5 = 50. Finally value of b is assigned to a += 50; a = a + 50 = 10 +50 =60.

The output of the above program is

a= 60, b= 50, c = 5, d = 10.

Table 1.10: Precedence and Associativity among C operators

Operator	Description	Associativity	Rank
( )	Function call	Left to right	1
[ ]	Array element reference		
-	Unary minus	Right to left	2
++	Increment		
--	Decrement		
!	Logical NOT		
~	Bitwise complement (1's complement)		
*	Pointer reference (indirection)		
&	Address		
sizeof( )	Size of an operand		
(type cast)	Type cast conversion		

*	Multiplication	Left to right	3
/	Division		
%	Modulus		
+	Addition	Left to right	4
-	Subtraction		
<<	Left shift	Left to right	5
>>	Right shift		
<	Less than	Left to right	6
<=	Less than or equal to		
>	Greater than		
>=	Greater than or equal to		
==	Equality	Left to right	7
!=	Inequality		
&	Bitwise AND	Left to right	8
^	Bitwise XOR	Left to right	9
|	Bitwise OR	Left to right	10
&&	Logical AND	Left to right	11
||	Logical OR	Left to right	12
?:	Conditional (or ternary) expression		13
=	Assignment operators	Right to left	14
*=			
/=			
%=			
+=			
-=			
&=			
^=			
|=			
<<=			
>>=			
,	Comma	Right to left	15

1.11 LVALUE AND RVALUE

An **lvalue** is an expression to which a value can be assigned. The lvalue expression is placed on the *left side*of an assignment statement, whereas an **rvalue** is located onthe *right side* of an assignmentstatement. Each assignment statement **must** have an lvalue and an rvalue. The lvalue expression **must**reference a storable variable in memory. It **cannot** be a constant. For instance, the following lines show a fewexamples of lvalues:

```
int x;

x = 1;
```

The variable x is an integer, which is a storable location in memory. Therefore, the statement x = 1 qualifies**x** to be an **lvalue**.

We have stated that an lvalue was defined as an expression to which a value can be assigned. It was also explainedthat an lvalue appears on the *left side* of an assignment statement. Therefore, an **rvalue** can be defined as anexpression that can be assigned to an lvalue. The rvalue appears on the *right side* of an assignment statement.

An **rvalue** can be a **constant** or an **expression**, as shown here:

```
int x, y;
x = 10;                  /* 10 is an rvalue; x is an lvalue */
y = (x + 5);             /* (x + 5) is an rvalue; y is an lvalue */
```

An assignment statement must have both an lvalue and an rvalue.

1.12 TYPE CASTING: PROMOTION AND DEMOTION OF VARIABLE TYPES

Up to this point, we have assumed that all of our expressions involved data of the same type. But, what happens when we write an expression that involves two different data types, such as multiplying an integer and a floating-point number? To perform these evaluations, one of the types must be converted.'C' provides two ways of type conversion:

1. **Implicit type conversion:** If operands are of different types, the **lower** type is automatically converted to the **higher** type before the operation carried out. The result of conversion is of higher type. The automatic type conversion process is called as **promotion** of data type to higher data type. The following sequence rules are followed while evaluating expressions.

(a) **char** data type is automatically type converted to **short** (or **shortint**).

(b) **short** is automatically converted to **int**, similarly **int** to **long, long** to **int long**.

(c) **float** is automatically converted to **double.**

(d) If one of the operand is **double**, the other is converted to **double**, and the result is **double**.

(e) If one of the operand is **unsigned**, the other is converted to **unsigned**, and the result is **unsigned**.

It is noted that **final** result of an expression is converted to the type of **lvalue** (i.e. variable on the left of assignment sign) before assigning the value to it. Yet, the subsequent changes are introduced during the final assignment. In this conversion process the following important aspects are considered:

(i) Conversion of **float to int** causes truncation of the fractional part.

(ii) Conversion of **float** causes rounding of digits.

(iii) Conversion of **long int** to **int** causes dropping of the excess higher order bits.

The above changes actually somewhat squeezes (i.e. truncation, rounding off) the **rvalue** and assign it to the **lvalue**. This process is known as **demotion**, because you are trying to fit a big data type object to the small data type object.

Look at the Program 1.9 that shows how an automatic promotion of variables takes place. Execute the program and verify the results.

```
/* Program 1.9 */
#include<stdio.h>
int main()
{       char c = 'a';
        float d = 234.5;
        int i = 354;
        short s = 45;
        printf("int * short is int: %d\n", i * s);
        printf("float * char is float: %f\n", d * c);
        d = d +c ;              /* char promoted to float */
        i = c + s ;             /* char and short promoted to int */
        printf("float + char : %f\n", d);
        printf("char + short is int:: %d\n", i);
        return 0;
}
```

The output of the following program is:

```
int * short is int      : 15930
float * char is float   : 15242.500000
float + char            : 299.500000
char + short is int     : 110
```

2. **Explicit type conversion:** There are many situations where we are forcing a type conversion in a way that is entirely different from automatic type conversion process. Suppose we need to calculate the speed of a vehicle, given that distance to be covered in 125km and time to cover the distance is 3 hours. The speed of vehicle is given by

Speed = distance/time

Since both, **distance** and **time**, are declared as integer values, the decimal part of the result of the division may be lost and the resulting **speed** value will represent a wrong figure. This problem can be overcome by converting one of the variables to the floating point as shown below:

Speed = **(float)** distance / time

The operator **(float)** converts the speed to the floating point number for the purpose of evaluation of the expression. Now, automatic type conversion plays its role and division is performed in floating point mode. Thus restoring the fractional part of computation. The use of operator**(float)**does not change the value of the variable **speed**. The value is changed locally within the expression. Such an **explicit type conversion** process is known as **casting**.

A **cast** tells the compiler to temporarily treat one data type as though it wasanother.

The general format of a **casting**is:

(type-name) expression

where**type-name** is one of the standard 'C' data types. The expression may be a **constant**, **variable** or an **expression**.

The **cast** has a **high** precedence and will be only applied to the **first** operand of anexpression, unless the whole expression is bracketed. If a cast is applied to a longer type then bits will be lost. A cast isequivalent to assigning the expression to a variable of the type and then using thatvariable in place of the whole construction.

Execute the program 1.10 and verify the results.

```
/* Program 1.10   */
#include<stdio.h>
int main()
{
{
int a = 2;
float x = 17.1, y = 8.95, z;
char c;
  c = (char)a + (char)x;
printf(" char c is    %c    :\n", c);
  c = (char)(a + (int)x);
printf(" char c is    %c    :\n", c);
  c = (char)(a + x);
printf(" char c is    %c    :\n", c);
  c = a + x;
printf(" char c is    %c    :\n", c);
  z = (float)((int)x * (int)y);
printf(" float z is    %f    :\n", z);
  z = (float)((int)(x * y));
printf(" float z is    %f    :\n", z);
  z = x * y;
printf(" float z is    %f    :\n", z);
        return 0;
}
```

Check your progress 2

1. State the output of the following program:

```
#include<stdio.h>
int main()
{
        int x = 7, y = -7, z = 11, w = -11, s =9, t = 10;
        printf(" x= 5d, y = %d, z = %d, w = %d, s = %d, t = %d\n"), x, y,
                z,w, s, t);
}
```

2. Assume i = 36 and j = 19

   What will the ouput of the following statement

   k -= (i> 15 && i <=234) ? –1 : i *j;

3. Write a C program to convert given number of days to a measure of time given in years, weeks and days. For example, 365 days is equal to 1 year, 1 week and thre days. (ignore leap year).

4. Write a progam which reads two integer numbers from keyboard and perform multiplication on them using bitwise operation.

5. Write a program to sum the digits of a four digit number.

6. Assume that *i, j* and *k* are integer variables and their values are 8, 5 and 0 respectively. What will be the values of variables *i* and *k* after executing the following expressions?

   (a) k = (j> 5) ? (i< 5) ? i – j : j – i : k - j;

   (b) i -= (k) ? (i) ? (j) : (i) : (k);

1.13 SUMMARY

This unit discusses various 'C' operators and their categories. 'C' expressions are valid combinations of operators and operands those compute a value determined by the precedence and associativity of the operators. The **weak** typing of C supports the **mixing** of operands of differing types. Operands may undergo type conversion before an expression takes on its final value. The general **rule** is that the final result will be in the highest precision possible given the data types of the operands. This may result in a promotion or a demotion, by which values are converted to wider or less wide types. A demotion may cause lost of most significant bits of an operand.

UNIT 2: DECISION STRUCTURES IN 'C'

Structure

2.0 INTRODUCTION

A computer program is a set of instructions for a computer. These instructions, also known as statements, are executed sequentially i.e., one after the other as they appear in a program. There are many situations in which the order of execution of these statements need to change based on certain specific condition or repeat a group of statements until specific conditions are fulfilled. This require a sort of decision making to check whether a particular condition is satisfied or not and then instruct the computer to execute particular instructions accordingly. A part of data is called **logical** if it conveys the idea of TRUE or FALSE. In real world life, **logicaldata** (TRUE or FALSE) are created in answer to a question that desires a **yes** or **no** answer. In computer science terminology, we **donot** use yes or no, we use TRUE or FALSE. In C language, FALSE is 0 and TRUE is anything other than 0, but most commonly 1.

In other words, we can say that a program can be much more powerful if you control the order in whichstatements are running. There are many situations in which we need to make selection between optional sections of a program. Suppose you are required to prepare two lists of candidates as given below:

1. List 1 contains name of students who will appear in examination.

2. List 2 contains name of those students who are not eligible to appear in examination.

These lists can be prepared on the basis of percentage of attendance of student. A student can appear in an examination if it has more than 75 % attendance, otherwise he or she will not eligible to appear in an examination. Therfore to decide whether a student will be appearing in examination or not, you need to compare his or her percentage of attendance with 75, and then take a decision about what to do next. So the decision statements are:

If attendance of student is greater than 75 then
Student is eligible for appear in an examination
Otherwise
student is not eligible for appear in an examination

Let us consider another problem where decision is to be made:

"A person is eligible for a job if his age is between 18 and 30 years"

The above sentence can be written in term of logical expression as:

age>=18 && age <=30

Remember that, we already discussed such kind of logical expression in previous Unit. Therefore, a person will be eligible only if this expression is evaluated to TRUE, otherwise the person will not eligible. Let us check this expression for a person having age 16.

16 >=18 &&16 <=30 =>FALSE&& TRUE =FALSE

Consider another statement:

"Ravi will join the company if either kamlesh is President or Rajul is Vice president"

The above statement is TRUE only when either Kamlesh is president or Rajul is vice president.

So, we can come to conclusion that **FALSE&& anything** is always **FALSE**, **TRUE | | anything** isalways true. It is noted that both the above problems consist of relational operator as well as logical operators. So in many situations we need to take decisions based on some logical data.

In this Unit, we discuss various decision (or selection) structures, including **if**, **if ... else**, **nested** if ... else, **switch** and **goto** that controls the flow of execution of a program.

2.1 OBJECTIVES

After working though this Unit, you will be able to:

- Know the simple **if** statement, **if... else** and **nested if .. else** construct.
- Understand when switch statement is used in programs
- Use goto statement for branching
- Write programs that involves simple to complex decision structures.

2.2 THE if STATEMENT

In 'C', **if** is a decision making statement and it is used to control the flow of execution of statements.The decision is described to the computer as a **conditional** statement that can be answered either true or false. If the answer is true, one or more action statements are executed. If the answer is false, then a different action or set of actions are executed. The **generalform** of a simple **if** statement is:

```
if (test-expression)
{
        Set-of-statements
}
        rest-of-statement-block
```

The *Set-of-statements* may be a single statement or a group of statements. If the *test-expression* is evaluated as true, the *Set-of-statements*will execute, otherwise *Set-of-statements* will skip and the execution transfers *to rest-of-statement-block*. It should note that if text-expression is true, then both *Set-of-statements*and *rest-of-statement-block* are executed in sequence. The ***test-expression*** may represent a *relation expression*, a logical expression,a *numeric variable* or a *numeric constant*. The specified condition may be a simple condition or compound condition.

Consider the following program 2.1 that test whether a given number is greater than 10.

```
/* Program 2.1 */
#include <stdio.h>
void main()

        int num = 20; /* initialize num by 20*/
        if( num > 10)

        printf("The number is greater than 10\n");

        printf("The number is:%d",num);
```

2.3 THE if . . . . else STATEMENT

When there are two possible outcomes, we can use **if - else** statement. Remember that*Otherwise* is another way of saying *else* in English. But In 'C', we only use **else.** The **if ..else** statement is an extension of **if** statement. The general form of if... else statement is as follows:

```
        if (test-expression)
{
        True-Set-of-statements
}
        else
{
        False-Set-of-statements
}
        rest-of-statement-block
```

If the *test-expression* is evaluated as TRUE, the *True-Set-of statements*will be executed; otherwise *False-Set-of-statements* will b executed, followed by execution of *rest-of-statement-block*in sequence In both the cases, **anyone** of the blocks will execute.

Consider a problem to determine whether a given number i positive. Program 2.2 shows how this can be done with the help of **if** . **else** statement.

/* Program 2.2 */

```
#include <stdio.h>
void main()
{       int number;
        printf("enter any integer number");
        scanf("%d", &number);
        if (number >= 0)
        printf("The number is positive");
        else
        printf("The number is negative");
}
```

Question 1: Below the two statements are given. Verify that thei simplified statements are correct.

Original statement	**Simplified statement**
If(a != 0) statement	If(a) statement
If(a == 0) statement	If(! a) statement

Question 2: Write a program to check whether the input year is a lea year or not.

2.4 NESTED if . . . else STATEMENT

When there are more than two possible outcomes, we u several **else** and **if**'s in nested form as follows:

```
if (test-expression 1)
{
        if (test-expression 1)
{
        Statements 1
}
else
{
        Statements 2
}
}
else
{
        Statements3
}
rest-of-statement-block
```

We need to know that whether input number is positive, negative or zero. There are three possible outcomes, so we can use **nested if . . . else** statement to identify the number.

Consider the following example where we need to know that whether two input numbers are same or one number is greater than other. Following program 2.3 demonstrates this case.

```
/* Program 2.3 */
#include <stdio.h>
void main()
{       int num1, num2;
        scanf("enter two integer numbers");
        scanf("%d %d", &num1, &num2);
        if (num1 <= num2)  /*  test num1 is greater than num2  */
        if( num1 < num2 )
                printf("%d is less than %d", num1, num2);
        else
                printf("%d is equal to %d", num1, num2);
        else    /*  num2 is greater than num1  */
        printf("%d is greater than %d", num1, num2);
}
```

Note:elseis always paired with the most recent unpaired **if**.

2.5 else . . . if LADDER

It is the collection of **if** statements with association of **else** statements when we have to perform multi-path decisions. A multi-path decision is a chain of **if**s in which the statements associated with each **else** is an **if**. It has the following general form:

```
if (condition 1)
  statement 1;
      else if (condition 2)
        statement 2;
            else if (condition 3)
              statement 3;
                  else if (condition n)
                    statement n;
                  else
                    default-statement;
rest-of-statement-block
```

This type of construct is known as **else . . if** ladder. The conditions are evaluated from top of the ladder to downwards. If any one of the condition evaluates to TRUE than all the statements in that block will executed.Execution pointer will directly jump to the *rest-of-statement-block* immediately after the closing curly braces. If all the given conditions evaluates to false than all the statements in else block will execute. Here, **else** block is also known as default statement.

The following example program takes a candidate's age and test score, and reports whether the candidate has passed the test. It uses the following criteria: candidates between 0 and 14 years old have a pass mark of 50%, 15 and 16 year olds have a pass mark of 55%, over 16's have a pass mark of 60%.

```
/* Program 2.4 */
#include<stdio.h>
void main()
{
        int age, score;
        printf("Enter the candidate's age: ");
        scanf("%d", &age);
        printf("Enter the candidate's score: ");
        scanf("%d", &score);
        if (age <= 14 && score >= 50)
{       printf("candidate age is %d and he passed the test.\n", age);
        printf("his score is %d\n", score);
}
        else if (age <= 16 && score >= 55)
{       printf("candidate age is %d and he passed the test.\n", age);

        printf("his score is %d\n", score);
}
        else if (score >= 60)
{       printf("candidate age is %d and he passed the test.\n", age);

        printf("his score is %d\n", score);
}
else
        printf( "This candidate failed the test.\n");
}
```

Question 3: Write a program that takes integer value in between 0 to 6 corresponding to Sunday to Saturday. The program will display name of the day according to the input number.

2.6 switch STATEMENT

We have seen that when multiple alternatives are given and only one is to be selected than use of the if statement can control the program. But the program becomes more complex when number of alternatives increases, not only for end user but also for the programmer who has designed that. 'C' provides us a multiple selection mechanism through the use of **switch** statement.

The switch statement is for the multi way decision. It is well structured, but it is having one**limitation** that it can be used only in certain **case**values where **onlyone** variable (or **expression**) is tested forcondition checking. When a match is found, a block of statements corresponding to that **case** is executed. **switch** statement can **only** test for equalitycondition (==).The general form of **switch** statement is shown in below:

```
switch (expression)
{
        case constant_1:
                group_of_statements_1;
                break;
        case constant_2:
                group_of_statements_2;
                break;
                :
                :
        default:
                default_group_of_statements;
                break;
}
rest-of-statement-block
```

All branches of the decision must depend on the value of that variable only. Also the variable used in condition checking must be an integer or character type (**int, long, short** or **char**).In switch-case statement each possible value of the variable can control a single branch and a final, catch all leftover cases. You should use **default** branch (final) for catching all unspecified cases.

The *expression* is an integer expression or characters. *constant_1, constant_2,* ....are constants or constant expressions (evaluate to an integer constant) and are known as *caselabels*. Each of these case labels must be *unique* with in a switch statement. The *group_of_statements_1, group_of_statements_2*... are the block of statements and may contain zero or more statements. Also, there is no need to put braces around these **group_of_statements**. However, each **case** label end with a colon (:).

When the control comes to switch statement, the value of **expression** is compared with each case labels, i.e. ***constant_1, constant_2***..... If a **case** is found whose value matches with the **expression** value, then the corresponding **group_of_statements**are executed sequentially.It should remember that **break** must be used after the end of every **group_of_statements**, which indicates the end of execution of block. Finally, control transfers to *rest-of-statement-block*. The **default** is an optional case. When you write it inside the switch statement, it will be executed when the value of expression does not match with any of the case values, otherwise no action is perfomed and control transfers to *rest-of-statement-block*.

Now, let us see the example below. This will clarify you the concept of *switchcase*.This example takes an objective which converts an integer into a text description or you can say that this program is working to estimate the number given as input.

```
int number;
/* Guess a number as none, one, two, many */
switch(number)
{case 0 :
        printf("The number is : None\n");
        break;
case 1 :
        printf("The number is :One\n");
        break;
case 2 :
        printf("The number is : Two\n");
        break;
case 3 :
case 4 :
case 5 :
default :
        printf("The number is : Many\n");
        break;
}
```

In the above example, each **case** is listed with some corresponding action.For examplecase 0 is associated with following action:

```
printf("The number is : None\n");
break;
```

The above switch statement uses empty cases. The **case 3, 4,** and 5and **default**execute the same statement.

Consider the following case, where weekdays are displayed as text strings on supply of digit in between 0 to 6 as input. This problem can also be implemented with the help of switch statement as shown below:

```
int day;
switch ( day )
{case 0: printf ("Sunday\n") ;
        break ;
case 1: printf ("Monday\n") ;
        break ;
case 2: printf ("Tuesday\n") ;
        break ;
case 3: printf ("Wednesday\n") ;
        break ;
case 4: printf ("Thursday\n") ;
        break ;
case 5: printf ("Friday\n") ;
        break ;
case 6: printf ("Saturday\n") ;
        break ;
default: printf ("Error — invalid day.\n") ;
        break ;
}
```

2.7 goto STATEMENT

The **goto** statement is used to alter the program execution sequence by transferring the control to some other part of the program. 'C' supports the **goto** statement to branch unconditionally from one point to another point in the program. The general syntax of **goto** statement is:

```
goto label;
```

where label is a valid 'C' identifier used to label the destination where the control could be transferred. A **label** has the same form as a variable name, and is followed by a **colon**. It can be attached toany statement in the same function as the goto. The scope of a **label** is the entire function. A label has following form:

```
label : statement;
```

There are a small number of situations where goto may find a place. The most common isto dispose of processing in some deeply nested structure, such as breaking out of two or moreloops at once. The **break** statement cannot be used directly since it only exits from theinnermost loop.Thus:

```
for ( ... )
        for ( ... ) {
                ...
                        if (failure)
                                goto fault;
                }
                ...
        fault:
                /* patch the fault */
```

This organization is helpful if the error-handling code is non-trivial, and if errors can occur inseveral places.

Consider the program 2.8 that illustrates the use of **goto** statement. The program computes the sum of first n natural numbers.

```
/*  Program 2.8 */
#include<stdio.h>
void main()
{
        intlimit, num, sum = 0;
        printf("Enter the limit for addition of natural numbers \n ");
        scanf("%d", &limit);
        num = 1;
target: sum += num;           /* target used as label    */
        if ( num < limit )
        {       num++;
                gototarget;
        }
        printf("Sum of first %d natural numbers is %d \n \n", limit, sum);
}
```

The output of the above program is:

```
Enter the limit for addition of natural numbers
10
Sum of first 10 natural numbers is 55
```

Check your progress 1

1. State the output of the following program

x =1 ;

```
y = 1;
if( n > 0)
        x =  y + 1;
        y = y – 1;
        printf(" %d %d", x, y);
```

What will be the values of x and y if n assumes a value of (a) 1 and (b) 0.

2. Write a program to find the type (i.e. equilateral, isosceles, Scalene or none) of triangle.

3. Write a program to find the number of and sum of all integers greater than 100 and less than 200 that are divisible by 7.

4. Write a program that will read the value of x and evaluate the following function

$$y = \begin{cases} 1 \; for \; x > 0 \\ 0 \; for \; x = 0 \\ -1 \; for \; x < 0 \end{cases}$$

Using (a) nested if statements
(b) Else if statements, and
(c) Conditional operator ?:

5. Create a program and execute it to check the category of the entered character.

```
        #include<stdio.h>void
        main()
{
        char ch;
        printf("enter the character\n");
         scanf("%c", ch);
         if( ch > = '0' && ch <= '9')
                printf("\n%c is a digit\n", ch);
        else if (( ch > = 'A' &&ch <= 'Z') | | ( ch >='a' && ch <= 'z'))
        printf("\n%c is an alphabet\n", ch);
        else
                printf("\n%c is a special charcatert\n", ch);
        }
```

6. Write a program to find the smallest of 3 numbers.

2.8 SUMMARY

The logical operators, like &&, || and !, relational operators can be sued in conjunction with if... else statements to create programs with branches. 'C' does not support data type for booelan decision. However, it can be cleverly implemented with the help of TRUE (anything) or FALSE (0). This unit discusses nested if..else construct, ladder else .. if and its better alternative switch statement. The unconditional branch statement goto can be used in few places; like, to get rid out from innermost loops.

UNIT 3: LOOP STRUCTURES IN 'C'

Structure

3.0 INTRODUCTION

In general statements are executed sequentially i.e the firststatement in a function is executed first, followed by the second, and soon.Most algorithms require a control structure that will allow us to repeat certain lines of code until a condition is reached. We call these **repeating** structures **loops**. 'C'provides various control structures that allow for more complicated execution paths. Loop structures are often desirable in coding in any language to have the ability to repeat a block of statements a number of times. In 'C', there are statements that allow iteration of this type. Specifically, there are two classes of program loops i.e. unconditional and conditional. An **unconditional loop** (for example, **goto**) is repeated a set number of times. In a **conditional loop** the iterations are halted when a certain condition is true. Thus the actual number of iterations performed can vary each time the loop is executed. The 'C' programming language has several structures for looping and conditional branching.

A loopstatement allows us to execute a statement or group of statements multiple times. Following is the general from of a loop statement (see Figure 3.1)which is found in most of the programming languages.

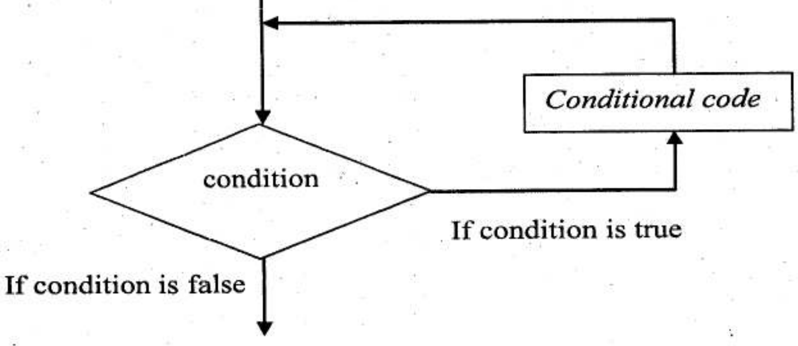


Figure 3.1: general form of loop statement

A loop is a sequence of statements those are executed until some conditions for termination of the loop is not satisfied. A loop therefore consists of two parts; one is *body of the loop* (or **conditionalcode**) and another is *control statement*(or **condition***)* as shown in Figure 31. The control statement (or condition) tests certain conditions and then allows the repeated execution of the statements contained in the body of the loop.

With respect to the **position** of control statement in the loop, a control structure may be classified either as the *entry-controlled loop* or *exit –controlled loop*as specified in Figure 3.2.

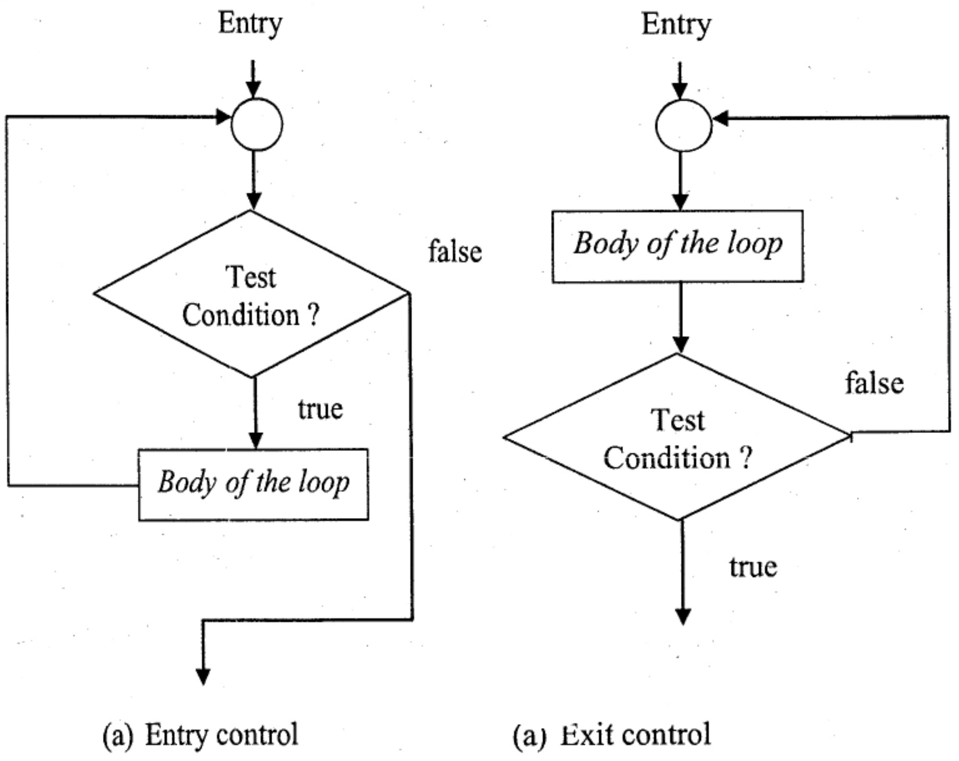


Figure 3.2: Loop control

In the ***entry-controlled-loop***, the control conditions are tested before starting of the loop execution. If the conditions are not satisfied, then the body of the loop will not be executed. In the case of ***exit-controlled-loop***, the control conditions are tested at the end of the body of the loop and therefore the body is executed unconditionally for the first time.

We can also distinguish two types of loops, which differ in the way in which the number of ***iterations***(i.e.,repetitions of the body of the loop) is determined:

- **Definite**(or **determinate**) **loops**, in which the number of iterations is known before we start theexecution of the body of the loop.The example of **definite** (or **determinate**) **loops** is the

for-loop that we will see in next section. The **definite** loop is also known as **counter controlled** loop.

Example: repeat for 10 times printing out a $.

- **Indefinite**(or **indeterminate**) **loops**, where the number of iterations is not known before we start to execute the body of the loop, but depends on when a certaincondition becomes true (and this depends on what happens in the body of the loop). This category has two types of "while" loops, **one** that allows the execution of the block **only** if a given condition is true and **another** that would allow the execution of the block **at least once**, and then, only if the condition is true. The **indefinite** loop is also known as **event- controlled** loop. An event-controlled loop will terminate when some **event** occurs. The event may be the occurrence of a **sentinel** value. A sentinel value is a special signal to indicate the end of input. There are other types of events that may occur, such as reaching the end of a data file.

  Example: while the user does not decide it is time to stop, take the print out and ask the user whether he wants to stop.

The 'C' language provides three types of loop structures for performing loop operations. They are:

(a) **while** statement

(b) **do.. while** statement

(c) **for** statement

3.1 OBJECTIVES

After working though this Unit, you should be able to:

- Write programs using **for, while**() and **do.. while**() loops.
- Use the **break** statement to jump from the loops.
- Use **continue** statement to abort the current iteration and start new iteration.

3.2 while STATEMENT

The **while** loop is the simplest loop structure.The **while** statemer allows us to perform pretest loops. Thus, it is an *entry controlled* loop The syntax (or format) of thewhile loop is as follows:

```
while (conditional-expression)
{
body of the loop
```

}

The while loop executes as long as the *conditional-expression* is true. The *body* of the while loop is a compound statement which is repeatedly executed and again conditional expression is evaluated. The *body of the loop* must have some statement which will eventually makes the condition false. The loop terminates when the conditional expression becomes false. If the conditional expression cannot become false, the while loop becomes an infinite loopexecuting endlessly. As the conditional-expression is evaluated first, the while loop statement will be executed **zero** or **more** times.

Every while loop will always containthree main elements:

- **Priming**: initialize your variables.
- **Testing**: test against some known condition.
- **Updating**: update the variable that is tested.

Consider a simple example

```
/* Program 3.1   */
#include <stdio.h>
#define MAX 5
voidmain ()
{
int index =1;                /* priming: initialization of variable        */
while (index <= MAX)         /* testing: test the condition             */
{
        printf ("Index: %d\n", index);
        index = index + 1;
        /* updating: increment the variable          */
}
}
```

The output of the above program is:

```
1
2
3
4
5
```

The above program initializes the variable index, test the value of index with some predefined value and then incrementing the index value by 1 inside the while loop. The updating of index is very essential otherwise test-condition never becomes false and while loop goes into an infinite loop.

Example:

A class teacher needs to calculate the average exam grade for a class of 10students. Such kind of problem can also be solved by using while loop. In this case, grade is the required input and average of grade is requested output. We initially assign variable counter as 1 and increment its value up to 10 (as maximum student is 10 in a class). On each iteration, we check whether counter reaches to 10. If it is not, the grade is taken as input and total grades are summed up on each iteration. With the body of loop, counter variable is incremented by 1. Thus, it is guaranteed that while loop will terminate after finite number of steps.

```
/*  Program 3.2   */
#include <stdio.h>
int main ( )
{
        int counter, grade, total, average ;
        total = 0 ;
        counter = 1 ;
        while ( counter <= 10 )
{
        printf ("Enter a grade : ") ;
        scanf ("%d", &grade) ;
total = total + grade ;
counter = counter + 1 ;
}
average = total / 10 ;
printf ("Class average is: %d \n", average) ;
return 0 ;
}
```

Question 1: The program 3.2 works only with class sizes of 10. What about if class size vary after few months? We would like our program to work with any class size.Hence, modify the above program which asks the user about how many students are in theclass. Use that number in the condition of the while loop and when computing the average.

The program 3.3 shows the use of while loop to check user input. If entered numer is negative, then appropriate message will appear, otherwise; input number is displayed on screen.

```
/*   Program 3.3   */
#include <stdio.h>
int main ( )
{
int number ;
printf ("Enter a positive integer : ") ;
scanf ("%d", &number) ;
while ( number <= 0 )
{
```

```
printf ("\nThat's incorrect. Try again.\n");
printf ("Enter a positive integer: ");
scanf ("%d", &number);
}
printf ("You entered: %d\n", number);
return 0;
}
```

We have already stated that sometimes while loop is worked as an **event controlled loop**. It is NOT known in advance exactly how many times a loop will execute. Consider the following program 3.4 which shows the sentinel value as an indication of end of input. The sentinel value serves as an event to occur. The program adds the value that is taken as input from keyboard. The input ends as soon as user type -1.

```
/*  Program 3.4  */
#include<stdio.h>
voidmain()
{
        sum = 0;                  /* initialization of variable  */
        printf("Enter an integer value: ");
        scanf("%d", &value);
        while ( value != -1)       /*   -1 is a sentinel value; it indicates
        end of input */
{
        sum = sum + value;  /* data modification   */
        printf("Enter another value: ");
        scanf("%d", &value);
}
}
```

The following program 3.5, calculates the mean and standard deviation. We have used sqrt function to find the square root. Since the function is from the math library we **must have the directive #include <math.h> before main ().** The mathematical library takes only arguments of type double. Hence we convert float to double by *explicit cast operator (double) before we* apply sqrt function. The mathematical library does not convert float to double by default. The program stops when user inputs the frequency f as 0. Verify the result on your machine.

```
/* Program 3.5   */
#include<stdio.h>
#include <math.h>
voidmain()
```

```
{       float x, mean, devi, sum=0, sqsum= 0;
        int n=0, f = 1;
        while ( f != 0)
{       printf("\nenter value & freqency");
        printf("enter freqency 0 to stop");
        scanf ("%f%d", &x, &f);
        n += f;            /*n is total number of observations*/
        sum += f*x; sqsum += f*x*x;
}
        mean = sum/n;
        devi = sqrt((sqsum / n - mean * mean));
        printf("\n total number = %d", n);
        printf("\n mean = %.2f", mean);
        printf("\n std. deviation = %.2f", devi);
}
```

3.3 **do . . . while** STATEMENT

The **do...while** statement is a variant of the **while** statement in which the condition test is performed at the "bottom" of the loop. The**do....while** statement is used to write **post-test** loops. Thus, it is an *exit controlled* loop. The **syntax** (or **format**) of the**do ...while** loop is as follows.

```
do
{
        body of the loop
}while (conditional-expression);
```

Note that, the **conditional-expression** appears at the end of the loop, so body of the loop execute once before the condition is tested. If the **conditional-expression** is **true**, the flow of control jumps back up to **do**, and **body of the loop** execute again. This process repeats until the given condition becomes **false**. As the conditional-expression is evaluated after the execution of body of the loop, this guarantees that the loop is executed **at least once**.The do loop terminates with a **semi-colon** (;).

The following program 3.6 reverses an integer with a **do.....while**loop. Check the output of the program:

```
/* Program 3.6 */
#include<stdio.h>
voidmain()
{
        int value, r_digit;
        printf("Enter the number to be reversed.\n");
        scanf("%d", &value);
                do {
                        r_digit = value % 10; /* obtain remainder  */
                        printf("%d", r_digit);/* print remainder  */
                        value = value / 10;      /* modify value; */
        }       while (value != 0);     /* test until value becomes zero */
        printf("\n");
}
```

The above program just displays the reverse of number, however it actually does not store that number. Modify the above program so that it stores the value of reverse number as well as display it.

3.4 for STATEMENT

It is simply a new way to describe the "**while**" loop. We use **for** loop in place of **while** when both start and end conditions are known and after every iteration of the loop there is an increment step. A **for** loop is a repetition control structurethat allows you to efficiently write a loop that needs to execute a specific number of times.The **for** loop is an example of **counter controlled** loop and **entry controlled** loop.

The basic syntax of the forstatement is:

for (***initialization expression;test expr;increment expr***)

```
{
        program statement(s);
}
```

The flow of control in a **for** loop is described below:

1. The ***initialization expression***step is executed first, and only **once**. This step allows you to declare and initialize any loop control variables. It is noted that you are **not** required to put a statement here, as long as asemicolon appears.

2. Next, the ***test expr*****(or condition)**is evaluated. If it is **true**, the **program statement(s)** inside the body of the loop is executed. If it is **false**, the **program statement(s)** inside the body of the loop does not execute and flow of control jumps to the next statement just after the **for** loop. The body of the loop is usually a compound statement. Hence, we have enclosed it in the curly braces. If it is a simple statement the braces are unnecessary.

3. After the ***program statement(s)*** inside body of the **for** loop executes, the flow of control jumps back up to the ***increment expr***

statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the condition.

4. The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (***program statement(s)***, then ***increment expr***, and then again ***test expr***). If the condition in test expression (**test expr**) becomes false, the **for** loop terminates.

Consider a simple example where a variable starts from initial value of 15 and stop when its value becomes 20. On every iteration, the current value is displayed.

```
/*  Program 3.7  */
#include <stdio.h>
int main ()
{
/* for loop execution */
for( int a = 15; a < 20; a = a + 1 )
{
printf("value of a: %d\n", a);
}
return 0;
}
```

The output of the above program is:

```
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

The for loop is easier to write and is equivalent to a while loop.

A loop becomes **infinite** loop if the test expression condition never becomes **false**. A**for** is normally used to make a loop infinite. Since **none** of the three expressions that form the **for loop** are required, you can make an endless loop by leaving the conditional expression empty. The following program is an example of infinite loop.

```
#include <stdio.h>
int main ()
{
        for( ; ; )
{
        printf("I am running forever.\n");
}
        return 0;
}
```

When the **conditional expression** is **absent**, it is assumed to be **true**. Using**for(;;)** construct signify an infinite loop.

NOTE: You can terminate an infinite loop by pressing **Ctrl + C** keys.

3.5 NESTED LOOPS IN C

'C' programming language allowsusing one loop inside another loop. Following section shows few examples to illustrate the concept. The syntax for a nested **for, while** and **do... while** loop statements in 'C'are as follows:

```
for ( init; condition; increment )
{
        for ( init; condition; increment )
        { statement(s);
        }
        statement(s);
}
while(condition)
{
        while(condition)
        { statement(s);
        }
        statement(s);
}
do
{
statement(s);
do
{       statement(s);
        }while( condition );
}while( condition );
```

You can place any type of loop inside of any other type of loop. For example a **for**loop can be inside a while loop or vice versa.

Consider the following program 3.8 that uses a nested for loop to find the prime numbers from 2 to 50.

```
/*  Program 3.8  */
#include <stdio.h>
int main ()
{
        /* local variable definition */
        int i, j;
        for( i =2;  i <50; i++)
        {
                for(j=2; j <= ( i /j); j++)
                if(!( i %j)) break;                // if factor found, not prime
                if(j> ( i /j)) printf("%d is prime\n", i);
        }
        return 0;
}
```

The output of the above program is:

```
2 is prime
3 is prime
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
23 is prime
29 is prime
31 is prime
37 is prime
41 is prime
43 is prime
47 is prime
```

3.6 FLOW CONTROL STATEMENTS

Flow control statements determine the next statement to execute. Thus, the statements which we have studied earlier like; **if-else**, **if,switch**, **while**, **for**, and **do** are flow control statements. But the above statements do not allow us to determinein an **arbitrary way** which is the next statement to be executed. As an alternative they *structure the program*, and theexecution flow is determined by the structure of the program.

'C'allows us to use *jump statements*.These statements are also flow control statements that cause the interruption of the execution flow and jumpsto a different statement that is not lie in the successive sequence path of program statements.

There are following two types of **Jump statements** are used in 'C':

- break (jump to the statement immediately following the current loop or switch statement)
- continue (jump to the condition of the loop)

3.6.1 break statement

This is a valuable statement when you need to jump out of a loop depending on the value of some results calculated in the loop. The break will jump out of the loop you are in and begin executing statements following the loop, effectively terminating the loop. The **break** statement can be used in **while**, **do-while**, and **for** loops to cause premature exit of the loop.

It has following two usages:

1. When the **break** statement is encountered inside a loop, the loop is immediately terminated and program control resumes at the next statement following the loop.
2. It can be used to terminate a case in the **switch** statement (already covered in decision making).

If you are using **nested loops** (i.e. one loop inside another loop), the **break** statement will stop the execution of the innermost loop and start executing the next line of code after the block.

The syntax of a **break** statement is as:

break ;

This statement has to be placed inside the **body** (a block usually)of a ***while, do..while***or ***for*** statement. It stops the execution of the bodystatement and jumps to the end of the statement. The remaining statementsof the including block are simply skipped. The *break* statement is usuallyassociated with an ***if*** statement to decide if the loop has to be exitedor not.

Consider the following program to illustrate the working of break.

```
1.    voidmain()
2.    {
3.              int i;
4.              for (i = 0; i < 5; ++i)
5.              {
6.                        if ( i == 3)
7.                                  break;
8.                        printf(" %d ",i);
9.              }
10.             printf("value of i after break statement is %d", i);
11.   }
```

The output of first, second and third iteration is 0, 1, and 2 respectively. When i is 3, we encounter the break statement. The control is transfer to the statement just following the exited loop, which is line number 10. So output "value of i after break statement is 3" is displayed.

When several loop statements are embedded together, a ***break***statement will be applied to the **closest** loop statement:

```
for (i = 0; i < 10; ++i)
{
        while ( i != 5)
        {
                if ( i == 2 )
                break;
        printf(" %d ",i);
        }
        printf(" %d ",i);
}
```

In this example, the ***break statement*** will stop the **while** loop only when I becomes 2.

3.6.2 continuestatement

The **continue** statement can be used in **while,do-while**, and **for** loops, and its effect causes the remaining statements in the body of the loop to be skipped for the current iteration of the loop. The syntax for a **continue** statement in 'C' is as follows:

continue;

Example: Print out the odd numbers between 0 and 100.

```
for (int i = 0; i <= 100; i++) {
        if (i % 2 == 0)
                continue;
        printf("%d", i);
}
```

For the **for loop**, **continue** statement causes the conditional test and increment portions of the loop to execute. For the **while** and **do...while** loops, **continue** statement causes the program control passes to the conditional tests.

Normally, a **break** or a **continue** statement causes the exit from a single level of nesting of **switch** or **loop**statements in which it appears. However, such statements allow also for exiting from more than one level ofnesting of a switch or loop.Hence to do this, the statements that define a block can have a label:

label :loop-statement ;

A *label* must be a constant expression (analogous to those used in the cases of a switch statement).The statement is as:

break*label* ;

interrupts the loop that has the label specified in the break statement. If there is no loop with the specifiedlabel that surrounds the **break***label* statement, then a compile-time error is signaled.Consider a Program 3.9 that causes exiting from more than one level ofnesting of **for** loop.

```
/* Program 3.9 */
#include<stdio.h>
main()
{
        for ( i = 1; i <15; i = i + 1 )
        {
                for ( j = 1; j <13; j = j + 1 )
        {
                if ( j % 7 == 0 )
        {
                break out;      /* use of break label;  statement */
        }
        else
        {
                printf ("X") ;
        }
        }
        printf ("\n") ;
}
out: printf(" I am in out label,exiting from all nested loops");      / * out
label */
}
```

3.7 SUMMARY

'C' provides three lop structures to control the repeated execution of one or more statements. The counter controlled; **for** loop is a counting loop that may also involve other conditional testing, used extensively with arrays and uses pretesting of the loop control variable(s). The **while** or **do-while** loop is used for event controlled repetition. A **while** loop is used when it is possible that the loop may never execute. It is a conditional loop that often doesn't involve counting, it uses pretesting of the loop control variable(s). A **do-while** loop is used when the loop must execute **at least** one time. It uses post-testing of the loop control variable(s) and generally used with interactive input. C allows us to use *jump statements*. These statements, like break and continue, are also flow control statements that cause the interruption of the execution flow and jumps to a different statement that is not lie in the successive sequence path of program statements.

Check your progress 1

1. Check the output of the program:

```
#include<stdio.h>
voidmain()
{
        int i=1; factorial=1;
        while (i<=n)
        {       factorial *= i;
                i=i+1;
        }
}
```

2. Will this loop end?
```
j=15;
while (j--);
```
3. find the output of the programs (a) to (h) after correcting syntax errors , if any.

(a)
```
void main()
{
int k;
        for ( k = 1; k <=8 ; k+=2)
        printf("%d", k*=3);
}
```
(b)
```
void main()
{
        int k = 0, p =1;
        for ( k = 10; k >0 ; —k)
        { p *= k++;
        k -=2;
        }
        printf("%d%d", k , p);
}
```
(c)
```
void main()
{       int k = 0,sum = 0;
        while ( k <=4)
        sum *= k++;
        printf("%d",sum);
}
```
(d)
```
void main()
{       int c;
        while (! (c = getch()) )
        putchar(c);
}
```
(e)
```
void main( )
{
        int k = 1;
        do
        {
                printf("%d", k);
        } while ( k > k);
}
```
(f)
```
void main()
{
        int k = 3, sum = 0;
        for ( k = 0; k <= 5; k++)
        {
        if (k > 3)
                continue;
        sum += k;
}       printf("%d %d", k, sum);
}
```

(g)

```
void main()
{
        int k = 3, sum = 0;
        for ( k = 0; k <= 5; k++)
        {       if (k > 3) break;
                 sum += k;
        }       printf("%d %d", k, sum);
}
```

(h)

```
void main()
{
        int x,y =2;
        for (x =5; x < 18; x += 4)
        {
                y += x;
                printf("%d %d\n",x,y);
        }

}
```

4. Read a series of 'n' numbers and add up them. Also find the squares of those numbers. After all numbers are input, print the *mean and standard deviation.*

5. How many times does **for** loop execute?

(a) `for(i = m; i <= n; i++)`

(b) `for(i = m; i < n; i++)`

(c) `for(i = m; i < n; i += x)`

6. The following program print every Nth number between 0 and 100, inclusively.

```
#include<stdio.h>
voidmain()
{
        const int N = 10;
        int i;
        for(i = 0;
        i <= 100;i  += N)
        printf("%d\n", i);
}
```

Answer the following questions.

(a) How many times is the loop executed?

(b) What is the value of the loop control variable **i** when the loop terminates?

7. Write code to output the first two columns:

```
0  1
1  2
2  4
3  8
4  16
5  32
6  64
```

8. Following program displays the conversion of temperature in Celsius to Fahrenheit.

```
#include<stdio.h>
voidmain()
{       int ctemp;
        float ftemp;
        printf("   ctemp  \t     ftemp");
        for (ctemp = 100; ctemp <= 0; ctemp--)
        {       ftemp = (9.0/5.0) * float(ctemp) + 32.0;
                printf ("%10.1f    %10.1f\n", ctemp, ftemp);
        }
}
```

Answer the following questions.

(a) What would happen if we had written (9/5) rather than (9.0/5.0)?

(b) What would happen if we had left off the parentheses from around (9.0/5.0)?

(c) Why was it a good idea to write float(ctemp) in the above expression rather than just ctemp?

(d) The loop as written will not execute at all. Why not, and how should you fix it?

9.

```
#include<stdio.h>
voidmain()
{
    for(i = 100; i >= 1; i--)
            printf("%d\n", i);
}
```

Answer the following questions.

a. What happens when test condition is changed to **i>1**? How about **i > 0**?

b. What is the value of the loop control variable **i** when the loop terminates?

10. Following program reads and adds a collection of positive floating point numbers (until a 0.0 value is read)

```
#include<stdio.h>
voidmain()
{
        float item;
        float sum = 0;          // initialize the sum
        printf ("Enter a positive number. Use 0 to stop)");
        scanf ("%f", &item);
        while (item != 0.0) // test for sentinel value 0.0
        {       sum += item; // sum = sum + item;
                 printf ("Enter a positive number. Use 0 to stop)");
                 scanf ("%f", & item);
        }
        printf ("The sum of the values just read is: %f", sum);
}
```

Answer the following questions.

(a) Re-write this loop to count the number of items read. Note that this number is not known before hand so it is more convenient to use a while loop here.

(b) Hand trace this loop to be sure it works. Use values 1.1, 2.2, and 3.3

(c) What will happen of the first printf /scanf pair is omitted? Trace the loop.

(d) What will happen if the second printf/scanf pair is omitted? Trace the loop.

(e) What would happen if there were just one pair of printf/scanf statements and these were placed at the very beginning of the loop?

11.

(a) What does the following code do?

```
        int x, y = 0;
        scanf("%d", &x);
while(x != 0) {
        y = (x % 10) + (y * 10);
        x /= 10;
}
        printf("%d", y);
```

Trace with x values 12, 452, and 1825. What about negative values?

(b) What does the following code do?

```
        int x, y;
        for(y = 0,
        printf("%d", x);                // expression 1
        x != 0;                         // expression 2
        y = x % 10 + y * 10, (x /= 10); // expression 3 and null body (the
                                                        semi-colon)
        printf("%d", y);
```

Trace with small x values 12, 452, and 1825.

(c) Are the two previous codes functionally equivalent?

12. Modify the program so that it does not make use of break statement.

```
#include <stdio.h>
int main ()
{
int a = 10;
while( a < 20 )
{
        printf("value of a: %d \n", a);
        a++;
                if( a > 15)
                {
                        break;
                }
        }
        return 0;
}
```

UNIT 4: ARRAYS AND STRINGS

Structure

4.0 INTRODUCTION

'C' programming language provides a data structure called the **array**, which can store a *fixed-size*(or *bounded*) *sequentialcollection* of elements of the **same type**. Arrays are an example of a **structured variable** in which (1) there are a number of elements of data contained in the variable name, and (2) there is an ordered method for extracting individual data items from the whole. An **array** is used to store the collection of data, but it is often more useful to think of an array as a collection of variables of the **sametype**.

Consider the case where you need to keep track of the roll number of student for a class of 100 students. Your first approach might be to create a specific variable for each student. This might look like

introll0 = 100   int roll1 = 101;   int roll2 =102;

It becomes increasingly more difficult to keep track of the roll numbers as the number of variables increase. Arrays offer a solution to this problem.

Instead of declaring individual variables, such as roll0, roll1, ..., and roll99, you declare **one array variable** such as **Roll No** and use RollNo[0], RollNo[1], and ..., RollNo[99] to represent **individual variables**. A specific element in an array is accessed by an **index number** or*subscript* in brackets after the array name. Thus,

RollNo[0] = 100,   • RollNo[1] = 101,   RollNo[2] = 102

represent individual variables with corresponding assigned values.

Individual variables are also called **elements** of an array.

All arrays consist of **contiguous** memory locations. The **lowest** address corresponds to the **first** element and the **highest** address to the **last** elementof the array as shown in Figure 4.1.

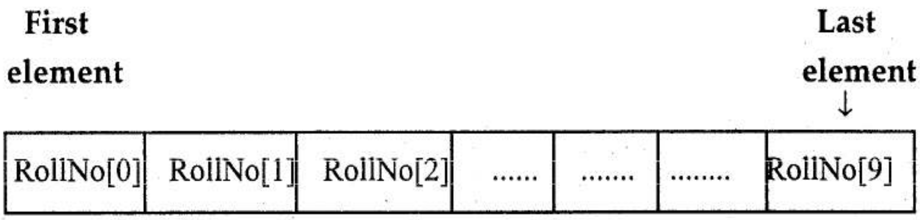


Figure 4.1 : Array consisting of 100 elements

Here we present some rules to create arrays:

1. All elements of an array are of the **same** type.
2. The number of elements in an array **cannot** change once you have instantiated thearray.
3. The type of the array can be **any** datatype, primitive or non-primitive (structure type).

4.1 OBJECTIVES

After working with this Unit, you should be able to:

- Declare and initialize one dimensional and multi dimensional array.
- Write programs using arrays of different data types.
- Apply various operations on an array.
- Know string arrays and various operations of them.

4.2 ONE DIMENSIONAL ARRAY

An array is a multi-element box (of **same** type), and uses an indexing system to find each variable stored within it. In 'C', indexing starts at **zero**. Arrays, like other variables in 'C', must be declared before they can be used.To declare an array in 'C', specify the **type** of the elements, **name** of an array and the **number** of elements in a **single subscript** [ ] as follows:

type arrayName [arraySize];

This is called a **single**-dimensional array. The **array Size** must be an integer constant greater than zero and **type** can be any valid 'C' data type. For example, to declare a 7-element array called **amount** of type **double**, use the statement:

double amount[7];

Here **amount** is a variable array which is sufficient to hold up-to 7 double numbers.Since **double** data type takes 8 bytes in a memory, an array variable occupies 8 * 7 = 56 bytes for the array in memory. During declaration **consecutive memory locations** are reserved for the array and all its elements as shown in Figure 4.2. Here, we assume that first element of **amount** array has physical address 4000. Since each element is double type, it takes 8 bytes in memory, so the next element starts at physical address 4008. Similarly, last element is at physical address 4048. After the declaration, you cannot assume that the elementshave been initialized to zero. There are random junk at each element's memory location.

The syntax for an element of an array items is ***amount[i]***, where *i*is called the **index** of the array element. The array element **amount[1]** is just like any normal double variable and can be treated as such. Each ***amount[i]*** element is also known as **logical address** of an array. In memory, one can picture the array **amount** as in the Figure 4.2.

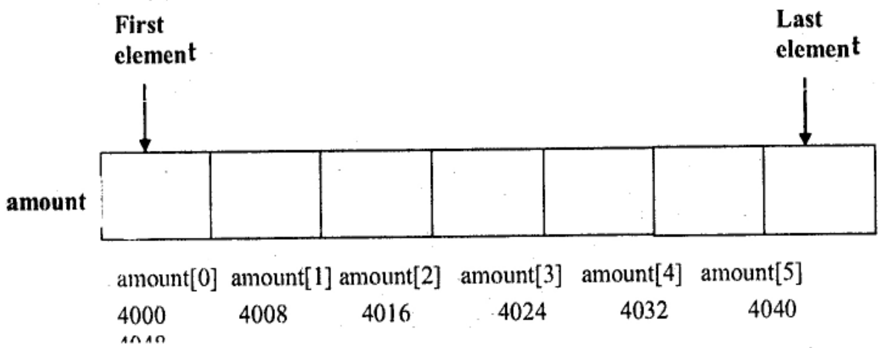


Figure 4.2: An array **amount** having 7 elements

Similarly, **int marks[100];** declares the **marks** as an array to contain a maximum of 100 integer constants.

Consider this fragment of program:

```
1:      int square[4];
2:       int i;
3:      for(i = 0;i <4; i++)
4:      square[i] = (i+1)*(i+1);
```

Line 1 defines an array with 4elements, each element is of integer type. The firstelement of an array has the index 0. Thus, this declaration gives 4 elements:

square[0]

square[1]

square[2]

square[3]

The last element has an index that is one less than the size of the array.

Line 2 defines a control variable for the loop that starts at **line 3** and iterates roundthe statement at line 4.

Line 4 in turn assigns to each element the square of its next position (that is one morethan its index).

In 'C' language, character strings can consider also an array of characters. The ***arraySize*** in a character string represents the maximum number of characters that the string can hold. For example,

char message[10];

declares the **message** as a character array (string) variable that can hold a maximum of 10 characters. Suppose we assign the **"HELLO"** string constant into the variable **message**. Each character of the string is treated as an element of the array message and is stored in the memory as follows:

t t

When the compiler sees a character string, it terminates it with an additional null character. Thus, the element **message[5]** stores the **null character '\0'** at the end. Thus during the declaration of arrays, we must always allow one extra character space for the null character.*It is unnecessary for the programmer to enter the null character, as 'C' adds it automatically.*

4.2.1 Initialization of Array

The general form to initialize the array is:

static data_type array_name[size] = { list of values };

where, values in the list are separated by commas. Note that, we have used the word **static** before declaration. This declares the variable as a static variable. If we want to initialize an array, then we can declare it as a static variable. This causes array elements initialized with 0.

You can initialize array in 'C' either one by one or using a single statement as follows:

double amount[7] = {1000.0, 200.0, 312.4, 1712.0, 50.0, 78.90, 45.67};

The number of values between braces { } cannot be larger than the number of elements that we declare for the array between square brackets [ ]

If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write:

double amount[] = {1000.0, 200.0, 312.4, 1712.0, 50.0, 78.90, 45.67};

You will create exactly the same array as you did in the previous example.

amount[6] = 50.0;

The above statement assigns 50.0 to the 7th element position in the array. Array with 6th index will be 7th i.e. last element because all arrays have 0 as the index for their first element which is also called as **base index** and the physical memory address for this first element is known as the **base address** of the array. Following is the pictorial representation of the same array we discussed above:

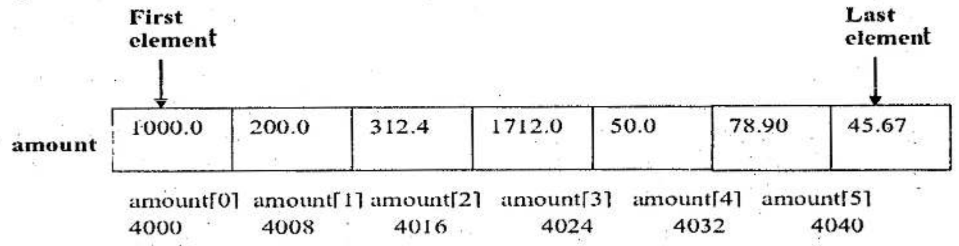


Figure 4.2: An array **amount** having 7 elements

You can also declare array element one-by-one using the following statements;

```
double amount[0] = {1000.0};
double amount[1] = {200.0};
double amount[2]= {312.4};
double amount[3]  = {1712.0};
double amount[4]  ={50.0};
double amount[5]  =  {78.90};
double amount[6] = {45.67};
```

If the declaration of an array is preceded by the word **static**, then the array canbe initialized at declaration. The initial values are enclosed in braces. e.g.,

static int value[9] = {1,2,3,4,5,6,7,8,9};

static float height[5]={6.0,7.3,2.2,3.6,19.8};

Some rules to remember when initializing during declaration:

1. If the list of initial elements is **shorter** than the number of array elements, theremaining elements are initialized to zero.

2. If a static array is **not initialized** at declaration manually, its elements areautomatically initialized to zero.

3. If a static array is declared **without a size** specification, its size equals thelength of the initialization list. In the following declaration, array **price** has size 5.

```
static int price[ ]={6,12,18,2,323};
```

As we have already discussed that indexing is the method of accessing individual array elements.Thus **msg[4]** refers to the **5th** element of the **msg** array. Acommon programming error is **out-of-bounds array indexing**. Consider thefollowing code:

```
int grade[3];
grade[5] = 78;
```

In the above example, array **grade** has only three elements. You are trying to access **5th** element which is actually **not** a part of array **grade**. The result of this **mistake** is unpredictable and machine, compilerdependent.

4.2.2 Accessing Array Elements

An element of array is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example:

```
double cost = amount[9];
```

The above statement will take 6th element from the array and assign the value to **cost** variable.

There is more suitable way to do the same thing. Remember that **array** variables and **for** loops often work hand-in-hand,since the **for** loop offers a convenient way to successively access array elements and perform some operation with them. Basically, the **for** loop counter can use to define the **index** for an array, as shown in the following summation example:

```
int total=0, i ;
int grade[4]={93,94,67,78};
    for ( i =0; i <4; ++i )
            total += grade[i];
```

The next Program 4.1 shows the use of all the above mentioned three concepts viz. **declaration**, **assignment** and **accessing** arrays:

```
/*  Program 4.1  */
#include <stdio.h>
int main ()
{       int n[ 10 ];            /* n is an array of 10 integers */
```

```
        inti ,j;
        /* initialize elements of array */
        for ( i = 0; i < 10; i++ )
        {
                n[ i ] = i + 50;  /* set element at location i to i + 50 */
        }
        /* output (or accessing )each array element's value */
                for (j = 0; j < 10; j++ )
        {
                printf("Element[%d] = %d \n", j, n[j] );
        }
        return 0;
}
```

The output of the above program is:

```
Element[0] = 50
Element[1] = 51
Element[2] = 52
Element[3] = 53
Element[4] = 54
Element[5] = 55
Element[6] = 56
Element[7] = 57
Element[8] = 58
Element[9] = 59
```

The above program declares an array **n** of size 10. The first **for** loop initializes all the 10 elements of an array **n** by a value (i + 50). The second **for** loop iterates all the 10 elements and produces the value of all the10 elements of an array **n**.

Consider a problem where we need to read the values from a one-dimensional array x having 10 elements and compute the sum of their squares. The program to solve this problem can see in Program 4.2 as follows

```
/*   Program 4.2   */
#include <stdio.h>
int main ()
{
        intx[ 10 ], sum =0;             /* x is an array of 10 integers */
        inti ;
        /*  read the input from the keyboard and stores the value
        in array element. */
        for ( i = 0; i < 10; i++ )
{   scanf("%d", &x[i]); /*  read input value for single array
        element  */
}
/* compute the sum of square of array elements */
for ( i = 0; i < 10; i++ )
{       sum  += x[i] * x[i] ;
}
/* display the result   */
printf("The sum of squares of array elements are %d", sum);
return 0;
}
```

Note that in the program 4.2, we take an input using scanf() and stores the received value into an array.

Consider a problem where we need to list the number of students those score the marks between 0-9, 10-19, etc. To do so, we use an array of 10 integers as **counters** for the ranges (See Program 4.3).

```
/*   Program 4.3   */
#include <stdio.h>
int main ()
{
int i,mark;
int totals[10];
/* initialize array with 0*/
        for (i=0; i<10; i++) {
                totals[i] = 0;
        }
/* read in the marks and add one to the relevant counter */
        printf("Enter marks finish with -1 \n");
        scanf("%d",&mark);
        while (mark != -1) {
                totals[mark/10]++;
                scanf("%d",&mark);
        }
/*print out the results */
printf("Number of students in the range \n");
for (i=0; i<10; i++) {
        printf("%d-%d is %d \n",
i*10,i*10+9,totals[i]);
}
```

Question 1: The program 4.3 will **fail** if a mark of 100 is entered. Modify the program so that it will work correctly when mark of 100 is entered.

4.3 TWO DIMENSIONAL ARRAY

The two dimensional arrays are useful in those situations where we need to store table of values or in matrix form i.e. row and column wise. A two-dimensional array is, in essence, a list of one-dimensional arrays. Each list itself consists of a one-dimensional array. The declaration of a 2D array can show as:

data_type arrayName [row_size][column_size];

Consider a 2D array element,

image[i][j]

the first index value **i** specifies a row index, while **j** specifies a column index.

Declaring 2D arrays is similar to the 1D array case:

int a[10]; /* declare 1D array */

float b[3][5]; /* declare 2D array */

Note that it is quite easy to allocate a large chunk of **consecutive memory** with 2D arrays. Array **b** contains **3x5=15 doubles** and since each double takes 8 bytes in memory. Thus, total 15*8 = 120 bytes in memory will allocate to this array.

A useful way to imagine a 2D array is as a **grid** or in**matrix**. Let us consider a declaration of array as:

int product[2][3];

This array will look like as:

product	0th column	1st column	2nd column
0th row	**product [0][0]**	**product [0][1]**	**product [0][2]**
1st row	**product [1][0]**	**product [1][1]**	**product [1][2]**

In 'C', **2D arrays are stored by row**. Which means that in memory the 0th row is put into its memory locations, the 1strow then takes up the next memory locations, and the 2nd row takes up the next memory locations, and so on. The Figure 4.3 shows the contiguous memory locations for allocation of array **product**. Since the array **product** is of **int** type, each element takes 2 bytes in memory. The total number of element in this array is row_size*column_size =2*3=6.

product[0][0] **product** [0][1] **product** [0][2] **product** [1][0] **product**[1][1] **product** [1][2]

The first element having the **logical** address **product**[0][0],**product**[0][1] for second element,**product**[0][2] for the third element in **firstrow**. In second row, **logical** address of elements are**product**[1][0],**product**[1][1] and **product**[1][2]. The physical addresses for each logical address are also shown just for understanding that memory locations are contiguous. The first element **product**[0][0] is stored at address 4002. The first row elements are stored from address 4002 to 4007. Similarly, second row elements are stored from address 4008 to 4013 as shown in Figure 4.3.

4.3.1 Initialization of 2D Array

The procedure of initializing the 2D Array is exactly similar to the initialization of 1D arrays. For example, consider the following array:

static int age[2][3]={4,8,12,19,6,5};/* array **age** with 2 rows and 3 columns */

The array is initialized row by row.

Thus, the above statement is equivalent to:

age[0][0]=4; age[0][1]=8; age[0][2]=12;/* for first row */

age[1][0]=19; age[1][1]=6; age[1][2]=5; /* for second row */

As before, if there are fewer initialization values than array elements, the remainder elements are initialized to zero.

To make your program more readable, you can explicitly put the values to be assigned to the **same** row in **innercurlybrackets** as shown below:

static int age[2][3]={{4,8,12},{19,6,5}};

In addition, if the number of rows is **omitted** from the actual declaration, it is set equal to the number of inner brace pairs as given below:

```
static int age[][3]={{4,8,12}, /* Note: row_size is missing */
                    {19,6,5}
               };
```

When the above statement is executed, compiler will set the row_size = 2. Commas are required after each inner brace that closes off a row, except in the case of the last row.

The inner brace pairs, which indicate the intended row, are **optional**. The following initialization is equivalent to previous example:

static int age[2][3]={ 4,8,12,19,6,5};

4.3.2 Accessing Two-Dimensional Array Elements

An element in 2-dimensional array is accessed by using the subscripts i.e. row index and column index of the array as:

int val = age[1][2];

The above statement assign array element **age[1][2]** to variable **val**. The variable **val**take 3rdelement from the 2ndrow of the array.

Again, as with 1D arrays, for loops and 2D arrays (or multi-dimensional arrays) often work hand-in-hand. Let us consider an example of program 4.4 as shown below, to access the elements of an array **age**:

```
/*  Program 4.4 */
#include <stdio.h>
        int main ()
{
        /* an array with 2 rows and 3 columns*/
                static int age[2][3]={{4, 8, 12}, {19, 6, 5}};
        int i, j;
        /* output each array element's value */
        for ( i = 0; i < 2; i++ )
        {
                for ( j = 0; j < 3; j++ )
                {
                        printf("age[%d][%d] = %d\n", i,j, age[i][j] );
                }
        }
        return 0;
}
```

The output of the above program is:

```
age[0][0]: 4
age [0][1]: 8
age [0][2]: 12
age [1][0]: 19
age [1][1]: 6
age [1][2]: 5
```

The above program uses two for loops to access array elements. Each loop is used for each subscript i.e. for row and column.

4.4 MULTI-DIMENSIONAL ARRAYS

'C' programming language allows **multidimensional arrays**. Multi-dimensional arrays have two or more index values which are used to specify a particular element in the array.The simplest form of the **multidimensional** array is the **two-dimensional** array. Here is the general form of a multidimensional array declaration:

```
data_type name[size1][size2]...[sizeN];
```

For example, the following declaration creates a three dimensional 5 .10 . 4 integer array:

```
int threedim[5][10][4];
```

When a number of arrays are to be declared using the same dimensions, it is goodpractice to use **#define** to set the bound. For example:

```
#define XDIM 10
#define YDIM 20
#define ZDIM 30
```

these can then be used in the program:

```
int testarray [XDIM][YDIM][ZDIM];
int i,j,k;
for (i=0;i<XDIM;i++)
for (j=0;j<YDIM;j++)
for (k=0;k<ZDIM;k++)
testarray[i][j][k]=i*YDIM*ZDIM+j*ZDIM+k;
```

Then if the program needs to cope with larger arrays it is only necessary to changethe definitions (and probably allow a longer run time and plenty of memory).

Check your Progress 1

1. What will be the output of the following program.

```
#include <stdio.h>
int main ()
{
    int values [12];
    for index;
    for (index=0; index<12;index++)
    values [index]= 2*(index+4);
    for (index=0; index<12;index++)
    printf(”The value at index= %2d is %3d\n”,
                    index, values[index];
    return 0;
}
```

2. Write a program to transpose a matrix.

3. A two dimensional matrix

$$A[4][5]=\begin{matrix} 2 & 3 & 5 & 1 & 0 \\ 2 & 5 & 7 & 3 & 45 \\ 12 & 37 & 56 & 0 & 23 \\ 2 & 13 & 58 & 36 & 71 \end{matrix}$$ is given..

Perform the following operations:

(a) Searching an element 45

(b) Sorting array elements in ascending order.

4. A and B are Two dimensional matrix

$$A[4][5]=\begin{matrix} 2 & 3 & 5 & 1 & 0 \\ 2 & 5 & 7 & 3 & 45 \\ 12 & 37 & 56 & 0 & 23 \\ 2 & 13 & 58 & 36 & 71 \end{matrix} \quad B[4][5]=\begin{matrix} 12 & 13 & 5 & 41 & 31 \\ 2 & 15 & 7 & 3 & 45 \\ 22 & 37 & 5 & 60 & 23 \\ 29 & 11 & 58 & 3 & 7 \end{matrix}$$

Perform the following operations:

(a) Addition of A and B

(b) Subtraction of A and B

(c) Multiplication of A and B

5. Write a program to check whether matrix A given in Question 4 is symmetric.

6. Write a program to find sum of even and odd numbers among n integers stored in an array.

7. A 1D array consist of 20 elements. Write a program to take input from keyboard and find the minimum and maximum valued element in an array.

8. Create a 2D array to store and print Pascal's triangle.

Sampleoutput:

```
1 1
1 2 1
1 3 3 1
1 4 6 4 1
```

4.5 STRINGS

A string is actually a one-dimensional array of characters which is terminated by a null character '\0'. Thus, a **null-terminated (orend-of-string character)** string contains the characters that comprise the string followed by a null. Don't forget to remember to count the end-of-string character when you calculate thesize of a string.Like the other'C' variables, strings must be declared before their use. Unlike other 1D arrays the **number of elements set** for a string set during declaration is only an **upper limit.** The actual strings used in the program can have fewer elements.The general form for **declaration**of a stringscan shown as:

static char array_name[size];

Consider the following code:

static char msg[18] = "Welcome";

The string called **msg** actually has only **8** elements. They are

'W' 'e' 'l' 'c' 'o' 'm' 'e' '\0'

String constants **marked with double quotes** andautomatically include the end-of-string character. The **curly braces are not required** for string initialization at declaration, but can be used if desired (**but remember the end-of-string character**).

The following declaration and initialization creates a string consisting the word **"Hello"**. To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word **"Hello"**.

char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};

If you follow the rule of array initialization then you can write the above statement as follows:

char greeting[] = "Hello";

Following is the memory presentation of above defined string in 'C'. Since string consists of char type (char data type takes 1 byte in memory), so this string takes 1*6 = 6 bytes in memory.Here, we assume that first element of array is at physical address 4000. Thus, last element which is **null** (or **end-of-string character**) is at physical address 4005 as shown below.

greeting	H	e	l	l	o	'\0'
	4000	4001	4002	4003	4004	

Actually, you do not place the null character at the end of a string constant. The C compiler automatically places the '\0' at the end of the string when it initializes the array. Let us try to print above mentioned string:

```
#include <stdio.h>
int main ()
{
        char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
        printf("Greeting message: %s\n", greeting );
        return 0;
```

}

The output of the above program is;

Greeting message: Hello

4.5.1 Initialization of strings

There are following of three ways to initialize the string:

(1) at the time of declaration,

char msg[18] = "Welcome";

(2) At the time of reading in a value for the string:

scanf("%s",name);

To readin a value for a string use the **%s** format identifier in **scanf**() statement.:

(3) At the time of using the **strcpy** function.

strcpy(msg, "hello"); Copies string **"hello"** into string **msg.**

Note: Direct initialization using the = operator is **invalid**. The following code would produce an error:

char name[34];

name = "Sameer"; / * I L L E G A L **INITIALIZATION*/**

Note: To initialize a string using **scanf** (See point 2 above), the address operator &is not needed for inputting a string variable. In 'C' an **array name is an address**. Infact, it is the **base address**of all the consecutive memory locations that makeup the entire array.

4.5.2 Accessing strings

The %s specifier is also used to display an array of characters that is terminated by the null ('\0') character. For example, the statement

prinf("%s", name);

can be used to display the entire content of the array **name.**

We have studied previously about the precision with which the array is displayed. For example, the specification

%10.5

indicates that the first five characters are to be printed in a field width of 10 columns. However, if we add minus sign in the specification (e.g., %-10.5s), the string will be printed left justified. The program 4.5 illustrates various effects of %s specifications.

```
/* Program 4.5 */
#include<stdio.h>
voidmain()
{
        static char greeting[15] = "Welcome Home";
        printf("\n\n");
        printf("————————————— \n");
        printf("|%15s|\n", greeting);
        printf("|%5s|\n", greeting);
        printf("|%15.6s|\n", greeting);
        printf("|%-15.6s|\n", greeting);
        printf("|%15.0s|\n", greeting);
        printf("|%.3s|\n", greeting);
        printf("|%s|\n", greeting);
        printf("————————————— \n");
}
```

The output of the above program is:

```
—————————
|   Welcome Home|
|   Welcome Home|
|            Welcom|
|Welcom              |
|                    |
|Wel|
|Welcome Home   |
—————————
```

The output illustrates the following features of the %s specifications.

(a) The entire string is printed when field width is less than the length of the string.

(b) The integer value (or precision) on the right side of the decimal point specifies the number of characters to be printed.

(c) Nothing is printed when the number of characters to be printed is specified as zero.

(d) If there is minus sign in the specification, then string is printed left justified.

The printf statement also support **variable field width** or **precision**. For example;

printf("%*.*s\n", w, d, string);

prints the first ***d*** characters of the string in the field width of ***w***.

4.6 STRING HANDLING FUNCTIONS

Almost every compiler comes with some standard predefined functions which are available for use. These are mostly input andoutput functions, character and string manipulation functions, and math functions. We will cover string handeling functions in this Unit. There is a standard 'C' library **<string.h>**that contains a number of useful string operations. The Table 4.1 shows some of them:

Table 4.1: String Functions available in standard <string.h> file

Category: String manipulation functions	
Functions	**Operations**
strlen	Finds length of a string
strcpy	Copies one string to another
strcmp	Compares two strings
strcmpi	Compares two, strings, non-case sensitive
strcat	Appends to a string
Category: String I/O functions	
gets(*string_name*);	reads in a string from the keyboard
puts(*string_name*);	displays a string on the monitor

Category: String manipulation functions

We shall discuss briefly how each of these functions can be used for the processing of strings.

(a) strlen() function:

This function counts the number of characters present in a string. While calculating the length; it doesn't count '\0'. The syntax for **strlen**() is:

```
len= strlen(string);
```

where, the variable **len** should be of int type because the **strlen** ( )function returns the number of characters in a **string** which is an **integer** value. The counting ends at the first null character.For example:

```
int n;

n =strlen("Hello World");
```

The value of n (or length of string) is: **11**

(b) strcpy() function:

The **strcpy function**is one of a set of built-in string handling functions available for the C programmer to use.The syntax of **strcpy** is:

```
strcpy(Target_string1, Source_string2);
```

when this function executes, **Source_string2** is copied into**Target_string1** at the beginning of **Target_string1**. The previous contents of **Target_string1** are overwritten. The **Source_string2** may be a character array variable or a string constant. The base address of the **Source_string2** and **Target_string1** should be supplied to this function. On supplying the base address, **strcpy ( )** goes on copying the characters in source string into the target string till it doesn't encounter the end of source string ('\0'). It is the responsibility of programmer to see that the target strings size is big enough to hold the sting being copied into it.In the following code, **strcpy** is used for string initialization:

```
#include <stdio.h>
#include <string.h>
voidmain ()
{
        char job[50];
        strcpy(job,"Interesting Language");
        printf("C is very %s \n",job);
}
```

The output of the above code is: **C is very Interesting Language**

(c) strcmp() function

This function compares two strings to find out whether they are **same** or **different**. The two strings are compared character by character **until** there is a mismatch or **end of one of the strings** is reached,whichever occurs first. If the two strings are **identical**, the function returns a value **zero**. If they are **not**,it returns the **numeric difference between the ASCII value of the first non-matching character**. The syntax for strcmp() is:

```
int var = strcmp(string1, string2);
```

where**var** is either 0 or numeric difference between the ASCII value of the first non-matching character.For example:

```
char   string1[ ]= "Apple";        /* string1 initialized to Apple */
char   string 2[ ]= "Applet";      /* string2 initialized to Applet */
inti, j, k, l, m;
i = strcmp(string1,string2);       /* both string differ at index 5
                                   "\0" and 't' are different
                                   character  */
j= strcmp(string1, "apple");       /* both string differ at index 0.
                                   'a' and 'A' are different
                                   character  */
k =strcmp( "Apple", "string2);     /* both string differ at index 5.
                                   "\0" and 't' are different
                                   character  */
```

```
l=strcmp("applet","applet");      /* both string differ at index 5.
                                     "\0" and 't' are different
                                     character  */

m= strcmp("apple","apple");       /* strings are same, so   m = 0  */
```

(d) strcmpi() function

strcmpi() compares *string1* and *string2* **without sensitivity** to **case**. All alphabetic characters in the two arguments *string1* and *string2* are converted to lowercase before the comparison. The general form is:

```
int var = strcmpi (string1, string2);
```

where **var** is either 0 or numeric difference between the ASCII value of the first non-matching character. The function operates on null-ended strings. The string arguments to the function are expected to contain a null character (\0) marking the end of the string.

This example uses strcmpi to compare the two strings.

```
#include <stdio.h>
#include <string.h>
int main(void)
{
 /* Compare two strings without regard to case */
 if (0 == strcmpi("hello", "HELLO"))
          printf("The strings are equivalent. \n");
else
        printf("The strings are not equivalent. \n");
return 0;
}
```

The output should be: **The strings are equivalent.**

(e) strcat() function

The strcat() function concatenates *string2* onto the end of *string1*, and returns *string1 to string3*. The syntax of this function is:

```
String3 = strcmpi(string1, string2);
```

For example:

```
        printf( "Enter your name: " );
        scanf( "%s", name );
        title = strcat( name, " the Great" );
        printf( "Hello, %s \n", title );
```

Category: String I/O functions

There are following two special functions designed specifically for string input and output:

gets(*string_name*);

puts(*string_name*);

The **gets** function reads a string from the keyboard. When the user hits acarriage return the string is inputted. The carriage return is not part of thestring and the **null** character (or **end-of-string character**) is automatically appended.

The function **puts** displays a string on the monitor. It does not print the **null** character (or **end-of-string character**).

Here is a sample program demonstrating the use of these functions:

```
charsentence[100];
printf("Please enter a sentence\n");
gets(sentence);
printf("You entered: ");
puts(sentence);
The output of the above program segment is:
        Please enter a sentence
        Accept this sentence. All the best.
        You entered:  Accept this sentence. All the best.
```

4.7 CHARACTER FUNCTIONS

There is a library of functions designedin 'C', to work with charactervariables. All of those function case are derived in **ctype.h** file, so whenever you are using any of these function, the ctype.h file has to be included on the top. The most commonly used functions are:

Table 4.2 : Limited character functions

Category: Testing a character

Return type: All the functions in this category test the **char**type variable **'ch'** and returns TRUE (1) or FALSE (0).

Function	**Operation**	**Example**

isalnum(ch)	Tests for alphanumeric character	char c;
		scanf("%c", &c);
		if(isalnum(c))
		printf("You enteralphanumeric char%c",c);

isalpha(ch)	Tests for alphabetic character.	char c;
		scanf("%c", &c);
		if(isalpha(c))
		printf("%c is a ASCII character\n", c);

isascii	Tests for ASCII character	char c='g';
		if(isascii(c))
		printf("You enter a letter of alphabet");

iscntrl	Tests for control character	Control characters are between 0 and 0x1F or equal to 0x7F

isdigit	Tests for 0 to 9	char c;
		scanf("%c", &c);
		if(isdigit(c))
		printf("You entered the digit %c\n", c);

isgraph	Tests for printable character	Same as isalnum(), isalpha(), iscntrl(), isdigit(), isprint(), ispunct().

islower	Tests for lowercase character	char c = 'a';
		if(islower(c))
		printf("%c is a lowercase letter\n", c);

isprint	Tests for printable character	Same as isalnum(), isalpha(), iscntrl(), isdigit(), isgraph(), ispunct(), isgraph().

ispunct	Tests for punctuation character	char c = ';';
		if(ispunct(c))
		printf("[%c] is a punctuation\n", c);

isspace	Tests for space character	char c = ' ';
		if(isspace(c))
		printf("[%c] is a space\n", c);

isupper	Tests for uppercase character	char c = 'A';
		if(isupper(c))
		printf("%c is a uppercase letter\n", c);

isxdigit	Tests for hexadecimal	Same as isalnum(), isalpha(), iscntrl(), isdigit(), isgraph(), ispunct(), isgraph().
Category: character conversion		**Return type:** depends upon chosen function

toascii	Converts character to ASCII code	char c ;
		c = toascii('A');
		printf("ASCII code is : %d \n", c);

tolower	Converts character to lowercase	char c ;
		c = tolwer('A');
		printf("lowercase is : %c \n", c);

toupper	Converts character to upper	char c ;
		c = toupper('a');
		printf("Uppercase is : %c \n", c);

In the following program 4.7, character functions are used to convert a string to all uppercase characters:

```
/*  Program 4.7  */
#include <stdio.h>
#include <ctype.h>
voidmain() {
        char name[80];
        int loop;
        printf ("Please type your name \n");
        gets(name);
        for (loop=0; name[loop] !=0; loop++)
        name[loop] = toupper(name[loop]);
        printf ("You are %s \n",name);
}
```

The output of the program is:

```
Please type your name
Sushant Singh
You are SUSHANT SINGH
```

4.8 SUMMARY

In this Unit we have studied one-dimension and multidimensional array, which are used to hold a group of variables of the same type. Each element of the array is identified and accessed by its subscript or position in the array. Array subscript begins at 0 for the first element. The array concept makes possible random access to any element. In classic 'C' arrays which must be initialized must be declared as **static**. A string is a group of characters usually letters of the alphabet. 'C' uses a string of data in some way, either to compare it with another string, output it, copy it to another string, or whatever, the functions are set up to do what they are called to do until a null, which is a zero, is detected. We have also studied some standard predefined functions which are available for use. These are mostly input/output functions, character and string manipulation functions.

Check your progress 2

1. Identify the missing statement.

```
mystrcpy(char dest[], char src[])
{
        int i = 0;
        while(src[i] != '\0')
        {
        dest[i] = src[i];
         ————        /*  missing statement  */
        }
        dest[i] = '\0';
        }
```

2. Write a function similar to strlen that can handle unterminated strings.

3. Is following two program fragments are equivalent?

```
printf ("Hello world\n");
char hello[] = {'H', 'e', 'l', 'l', 'o', ' ','w', 'o', 'r', 'l', 'd', '\n', 0};
```

4. Write a program that that converts all characters of an input string to upper case characters.

5. Write a function that returns true if an input string is a palindrome of each other. A palindrome is a word that reads the same backwards as it does forwards *e.g* ABBA.

6. Obtain the outputof the following program.

```
#include <stdio.h>
void main()
{
        int i, nc;



        nc = 0;
        i = getchar();
        while (i != EOF) {
        nc = nc + 1;
        i = getchar();
 }
        printf("Number of characters in file = %d\n", nc);
}
```

7. Obtain the output of the following program.

```
#include <stdio.h>
void main()
{
        int c, nc = 0;
        while ( (c = getchar()) != EOF ) nc++;
        printf("Number of characters in file = %d\n", nc);
}
```

Is this program is same as program given in question 7?

8. State what this program does?

```
#include <stdio.h>
void main()
{       int c, nc = 0, nl = 0;
        while ( (c = getchar()) != EOF )
{
        nc++;
        if (c == '\n') nl++;
}
        printf("Number of characters = %d, number of lines = %d\
        nc, nl);
}
```

9. What will be output if you execute the following 'C' code?

```
#include<stdio.h>
void main(){
        char arr[7]="Network";
        printf("%s",arr);
}
```

10. What will be output if you execute the following 'C'code?

```
#include<stdio.h>
void main(){
        char arr[11]="The African Queen";
        printf("%s",arr);
}
```

11. What will be output if you execute the'C'code?

```
#include<stdio.h>
void main(){
        char arr[20]="MysticRiver";
        printf("%d",sizeof(arr));
}
```

12. What will be output if you execute the following 'C'code?

```
#include<stdio.h>
#define var 3
void main(){
        char data[2][3][2]={0,1,2,3,4,5,6,7,8,9,10,11};
        printf("%o",data[0][2][1]);
}
```

13. What will be output if you execute the following 'C'code?

```
#include<stdio.h>
void main(){
        int arr[][3]={{1,2},{3,4,5},{5}};
        printf("%d %d %d",sizeof(arr),arr[0][2],arr[1][2]);
}
```

14. What will be output if you execute the following 'C'code?

```
#include<stdio.h>
void main(){
        int xxx[10]={5};
        printf("%d %d",xxx[1],xxx[9]);}
```

15. What will be output if you execute the following 'C' code?

```
#include<stdio.h>
void main(){
        long double a;
        signed char b;
        int arr[sizeof(!a+b)];
        printf("%d",sizeof(arr))
}
```

Suggested Further Reading

1. *The C Programming Language,* Kernighan, Brian W.; Dennis M. Ritchie *(February 1978), (1st ed.).* Englewood Cliffs, NJ: Prentice Hall.
2. Let us C, Yashwant Kanetkar, BPB, (2013)
3. Programming and Problem Solving Through "C" Language, Harsha Priya, R. Ranjeet, Firewall Media.
4. Computer Science: A Structured Programming Approach Using C (3rd Edition) 3rd Edition, Cengage Learning; 3 edition (February 6, 2006), Behrouz A. Forouzan (Author), Richard F. Gilberg
5. Programming Logic and Design, Joyce Farrel, Chegg publication.
6. Lecture notes on Programming Fundamentals, Giovanni Moretti, 1998.
7. Algorithms, flowcharts, data types and pseudocode, http://www.aves.ktu.edu.tr/
8. Program errors, www.hptunotes.com/
9. *C Printf and Scanf Reference, by Wayne Pollock, Tampa Florida USA*
10. *Principle of good programing, http://www.unaab.edu.ng*
11. *Program errors and exception handling, https://www.inf.unibz.it/*
12. *Introduction to Computer Programming Concepts, http://www.spotidoc.com*
13. *C Compilers Reference, CodeWarrior.*
14. *C programming tutorial, tutorialspoint.com*

U.P.Rajarshi Tandon Open University, Allahabad

UGCS-102
Problem Solving Through "C"

Block 4

Advanced Features of C

Block-4 INTRODUCTION

This block covers some advance features of 'C' language. The Unit 1 discusses the pointers and their relationship to arrays and character strings. Pointers are very useful part of 'C'and separate it from more conventional programming languages. Pointers make 'C' more influential allowing a wide variety of tasks to be accomplished. A pointer is a constant or variable that contains an address which can be used to access data. Pointers are built on the fundamental concept of pointer constants.

In Unit 2, we will see the concept of functions or modular programming. A program should be divided into a main module and its sub modules. Each module is then further divided into sub-modules (or pieces) until the resulting modules are basic. These smaller pieces sometimes called 'modules' or 'subroutines' or 'procedures' or functions. A 'C' program is made of one or more functions, one and only one of which must be named **main**. The execution of the program always starts with main, but it can call other functions, including library function such as **printf**() and **scanf**(), to do some part of the task. Generally, the purpose of a function is to receive zero or more pieces of data, operate on them, and return at most one piece of data. Call by reference is most efficient method as compared to call by value. Use of pointers makes the function programming most efficient.

In Unit 3, we will study how external files may read and write by 'C' programs. The file I/O functions are very similar to the console I/O functions, but high level I/O functions are very useful for handling large volume of data. The dynamic memory allocation functions are also explained with examples.

The Unit 4 describes the preprocessor directives which are the powerful tool for manipulating text files. While control directives, macro directives and conditional compilation are its most popular features. Being 'C' a system programming language, it provides you access at lower level in the form of return values. In case of any error, compiler sets an error code **errno** which is global variable and indicates an error occurred during any function call.

UNIT-1 Pointers

Structure

1.0 INTRODUCTION

This Unit starts with introduction and various manipulation operations apply on pointers. They are very simple but appear a little confusing for beginners. However, they are powerful and once you understand the basics, it is easy for you to write complex programs with great ease. The feature of pointers makes 'C' different from other programming languages. Some 'C' programming tasks are performed more easily with pointers, and other tasks, such as dynamic memory allocation, cannot be performed without using pointers. Hence it becomes necessary to learn pointers to become a perfect 'C' programmer. One thing which should be kept in minds that, "**pointers store addresses and NOT data values**". This is a very simple statement and completely covers the basics of pointers.

We have already discussed in previous Units that every variable has a **memory** location and every memory location has its **address** which can be accessed using **ampersand (&)** operator. Let us consider an example which will print the address of the variable:

```
#include <stdio.h>
int main ()
{
        int x;
        char y[.10];
        printf("Address of x variable: %x\n", &x );
        printf("Address of y variable: %x\n", &y );
        return 0;
}
```

The output of the program might be:

```
Address of x variable: b0a45400
Address of y variable: b0c523f6
```

Now, we have seen that what is memory address and how can we access it? Thus, the underlying framework is complete. Now, let us see what is a pointer?. In next subsequent sections we introduce little more intricacies about pointers.

1.1 OBJECTIVES

After going through this unit, you should be able to:

- After going through this unit, you should be able to:
- Understand and use of pointers;
- See the underlying unity of arrays, strings and pointers;
- Various arithmetic operations on pointers.
- Use address (&) operator and indirection (*) operator.

1.2 POINTERS AND ADDRESS OPERATOR

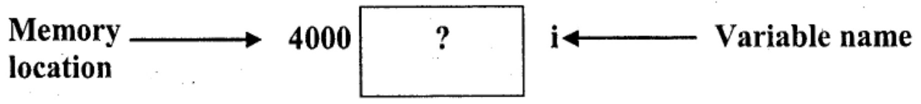
Pointers are very useful part of 'C'and separate it from more conventional programming languages. Pointers make 'C' more influential allowing a wide variety of tasks to be accomplished. Pointers enable us to:

- Reduce the complexity of program
- Increase the execution speed of program
- efficiently represent data tables
- alter values of actual arguments passed to functions ("call-by-reference")
- work with memory which has been dynamically allocated
- more concisely and efficiently deal with arrays and strings

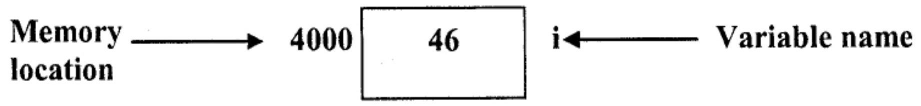
For the sake of simplicity, we assume that memory has capacity of 64KB, so its range is from 0 to 65, 535. When you declare a simple variable, like

int i;

a memory location with a certain address (say, 4000 is assigned by computer) is assigned to variable ***i*** so that any value can be placed in the variable. Thus we have the following picture:



After the statement i=**46;** the location corresponding to **i**will be filled



Remember that the **memorylocation** (i.e. 4000 in this example) is referred as **physical address** of variable *i*, whereas, **name** of variable (i.e. *i*, in this example) is known as **logical address**. Throughout the execution of program, this physical address is **bind**(or **attached**) to variable ***i***. Therefore you can **access** the variable **either** through its physical address **or** through logical address.

As we have already described that we can find out the **memory address of a variable** by simply using the address operator **&**. An **address expression**, which is also a **unary expression**, consists of an ampersand (**&**) and a variable **name**as follws:

&i

The above expression should be read as **"address of i"**, and it returns the memory address of the variable **i**.

The following program 1.1 demonstrates the difference between the content of a variable and its memory address:

```
/* Program 1.1 */
#include <stdio.h>
void main()
{
        float x;
        v=4.126;
        printf("The value of v is %f\n", v);
        printf("The address of v is %X\n",&v);
}
```

The output of the above program is:

```
The value of v is 4.126
The address of v is BFF0
```

The above program uses %X specifier, which displays the address in uppercase Hex code. Similarly, %x specifier displays lowercase Hex codes.

1.3 POINTER DECLARATION AND INITIALIZATION

A pointer is also a 'C' variable whose value is the address of another variable i.e. direct address of the memory location. Similar to other 'C' variables, pointers must be declared before they are used to store any variable address. The syntax for **pointer declaration**is as follows:

data_type *pointer_var_name;

Here, **data_type** is the pointer's base type; it must be a valid 'C' data type and **pointer_var_name** is the name of the pointer variable. The asterisk * used to declare a pointer is the same asterisk that you use for multiplication. However, in this statement the asterisk is being used to designate that the variable is a pointer type variable. Let us consider the following valid pointer declarations:

```
int *ip;                /* pointer to an integer */
double *dp;             /* pointer to a double */
float *fp;              /* pointer to a float */
char *ch                /* pointer to a character */
```

The *actual data type of the value of all pointers*, whether integer, float, character, or otherwise, is the same, a *long hexadecimal number that represents a memory address*. The only **difference** between pointers of different data types is the data type of the variable or constant that the pointer is referring to.

Now let us see how the memory representation of pointers looks like. Consider the following two statements:

Line 1 int age; /* declaration of variable: **age** is an integer variable*/
Line 2 ⟶ int *p; /* declaration of pointer variable : **p** is a pointer to an integer */

On executing Line 1, variable age gets some physical address from memory, say 4000.

Read the second Line 2 from right to left: **"p (* MEANS)is a pointer to an int"**

Once a pointer has been declared, it can be assigned an address. Assigning address of a variable to a pointer is referred as **initialization** of pointer variable. The general form for initializing pointer variable is:

pointer_var_name = &var_name;

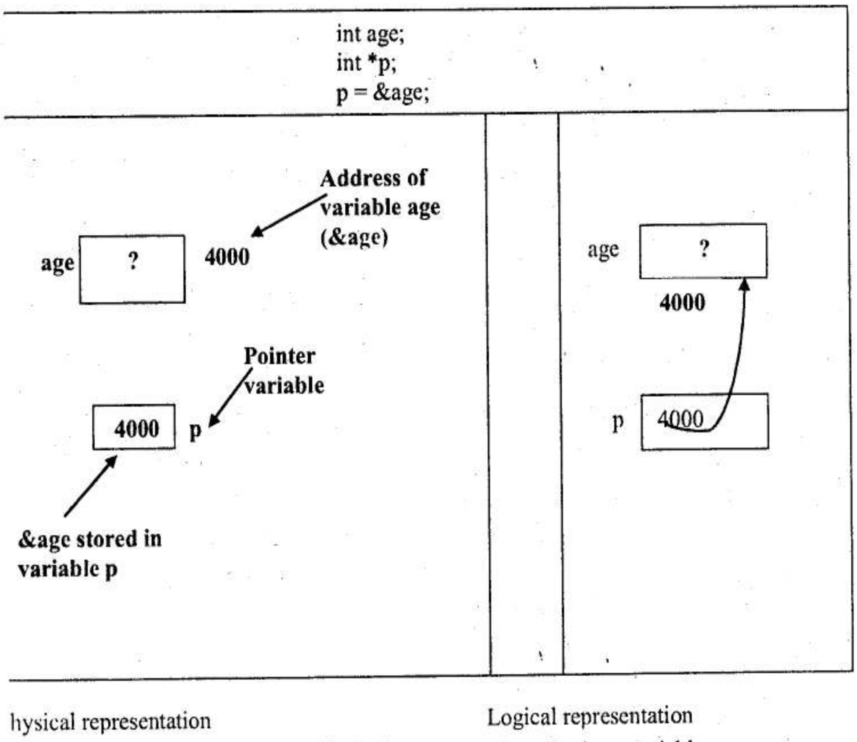
Now, assign the address of an integer variable **age** to a pointer variable **p**. The address is assigned using address operator (&) as.

Line 3 p = &age; /* p is assigned the address of **age** */

Read the third Line 3 from right to left: **"pis assigned the address of age"**, or

Read the third Line 3 from left to right: **"address of ageis assigned to pointer p".**

After this assignment, we say that **p** is "**referring to**" the variable **age** or "**pointing to**" the variable **age** (see logical representation in Figure 1.1**)**. The pointer **p** contains the physical memory address of the variable **age.** Since no value is assigned to variable **age,** the contents are represented as **?** in Figure 1.1.



```
int age;
int *p;
p = &age;
```

Figure 1.1: Memory and logical representation of pointer variable

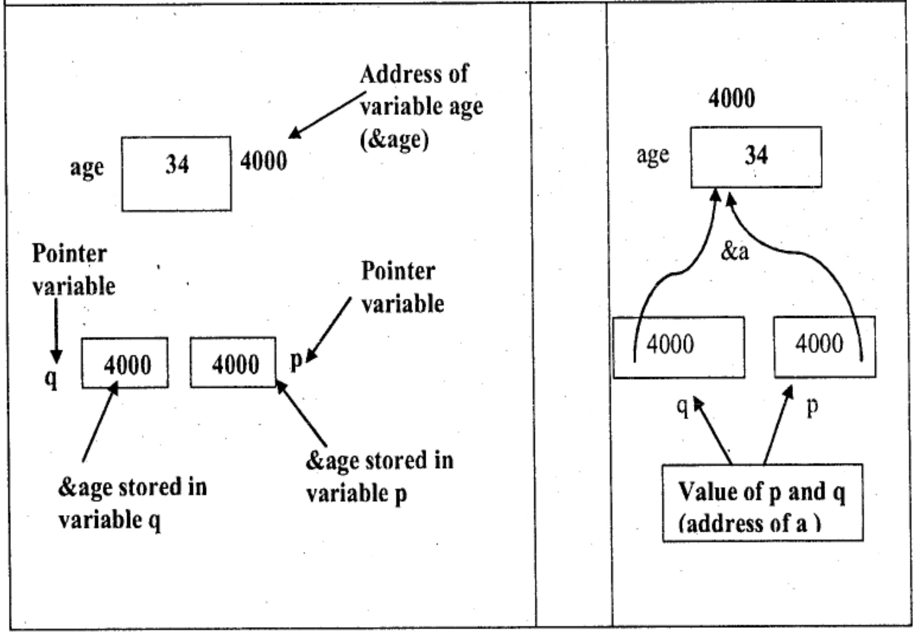
Hence, we have seen that how to declare and initialize a pointer variable and assigning the address of variable to a pointer. Now, let us consider the statement:

age =34;

When this statement is executed, the value 34 is assigned to a variable age. Or, value 34 is stored at physical address 4000. We have already discussed that the physical addresses are drawn from a set of memory locations and they are differ on each execution of program. Thus, pointer variable **p** points to an **int** variable **age** that has value 34 in it.

Let us see another example where multiple pointers point to same variable. Consider the statements shown in Figure 1.2.

```
int age = 34;          /* age is a  int variable having value 34   */
int *p = &age; /* p is a pointer to int age andaddress of ageis assigned to pointer p
int *q;          /* q is a pointer to int   */
q = p;  /*  q is assigned a value of p  MEANS address contained in p is assigned to q */
```

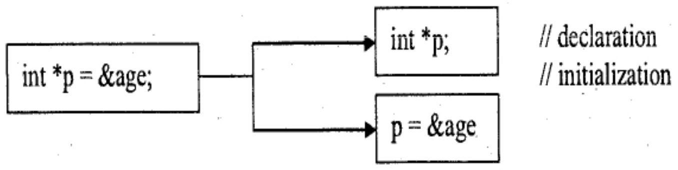


Address of variable age (&age)
age
34
4000
Pointer variable
q
4000
4000
p
Pointer variable
&age stored in variable q
&age stored in variable p
4000
age
34
&a
4000
4000
q
p
Value of p and q (address of a)

Physical representation Logical representation

Figure 1.2: Memory and logical representation of multiple pointer variables

The **first**statement declares an **int** type variable **age** with initial value of **34**. The **second** statement declares an **int** type pointer variable **p** pointing to address of variable **age**. The second statement can be **decomposed** into two parts as given below:



The **third** statement declares a pointer variable **q** of **int** type. In **fourth** line, we assign content of pointer **p** to pointer variable **q**. This **content** is actually the physical address of variable **age** (i.e. **4000**). Thus, **address4000** is assigned to pointer variable **q**. So, pointer **q** is now **points**

to (or **referring to**) variable **age**. Thus, **two** pointers **p** and **q** referring to **sameint** variable **age**.

Since pointers are also declared as variable, they also have memory address. Pointer **p** and **q** actually have their own memory addresses. The memory address is always positive constant, so content of pointer is always an unsigned positive value that represents address.

[Note]A pointer that points to no variable contains the special null-pointer constant, **NULL**.It is a good programming practice to assign a **NULL** value to a pointer variable in case you are not using that pointer. This can be done at the time of pointer variable declaration. A pointer that is assigned **NULL** is called a **null** pointer. The **NULL** pointer is a constant with a value of **zero** defined in several standard libraries of 'C'. Consider the following program 1.2:

```
/* Program 1.2   */
#include <stdio.h>
int main ()
{
        int *ptr = NULL;
        printf("The value of ptr is : %x\n", &ptr );
        return 0;
}
```

The output of the above program is: **The value of ptr is 0**

The memory address 0 has a particularmeaning. It indicates that the pointer is not deliberately pointing to an accessible memory location. Rather by convention, if a pointer contains the **null** (**zero**) value, it is assumed to point to **nothing**!

[End of Note]

One more thing, consider the following statements:

```
float price, amount;  /* declaration of float type variables  */
int  item, *ip;       /* declaring item as int variable, ip as pointer
                         variable */
p = &price;           /*   ILLEGAL ASSIGNMENT   */
```

The third statement assigns an **address** of **float** type variable to **int** type pointer variable, which result in **erroneous** output. You must **ensure** that pointer variables **always** point to the corresponding type of data. Thus, when you declare a pointer to be an **int** type, the system assumes that pointer will hold the address of variable which is also of **int** type.

1.4 INDIRECTION OPERATOR

There are some important operations which we can do with the help of pointers veryfrequently. **(a)** define a pointer variables **(b)** assign the address of a variable to a pointer and **(c)** finally **access** the value at the physical address which is available in the pointer variable. The first two points are already described in previous section. This section deals with third point, i.e. the **indirection** or **dereferencing**(*) operator.

The **unary** operator *, alsoknown as **indirection (or dereferencing)** operator**returns** the **valueof the variable** located at the address specified by its operand.The indirection operator is **complement** to the **address** operator. It is used as follows:

***p;**

The above expression is read as **"contents of p"**, or **"value at address pointed by p"**. The value stored at the memory address **p** is returned after executing this statement.

Let us consider the sample program 1.3:

```
/* Program 1.3  */
#include<stdio.h>
void main()
{
        int a= 2,b= 80,*ip;
        ip=&a;
        b=*ip;          /* equivalent to b=a */
        printf("The value of b is %d\n",b);
}
```

The output of the above program is: **The value of b is 2**

The statement **b = *ip** in above code can be read right to left as:

"value at address pointed by p is assigned to variable b"

The above code assign **indirectly** the value of variable **a** to variable **b** through the pointer **ip.**

Now let us have fun with pointers. Following program 1.4 accessesvariables through pointers.

```
/* Program 1.4  */
#include<stdio.h>
void main()
```

```
{
        int a= 4, *p= &a, *q=&a;      /*      Line 1  */
        a = a +4;                     /*      Line 2  */
        *p = 10;                      /*      Line 3  */
        *&a = *p + *q;                /*      Line 4  */
        a = *p * *q;                  /*      Line 5  */
}
```

The explanation of above code is given below:

Line 1: **a** is **int** variable. **p** and **q** are **int** type **pointers** containing the address of variable **a**.

Line 2: a is incremented by 4.

Line 3: *p = 10; means value at address pointed by p is set to 10.

Line 4: *&a = *p + *q;

First consider the lvalue expression, *&a

The precedence of & operator is higher than indirection operator, so **&a** is evaluated first. This refers to the statement **& a** as: **address of variable a**.

Next, ***(&a)** is evaluated, which means: **value at address (&a).**

Now consider the rvalue expression, *p + *q

*p + *q means: **value at address pointed by p + value at address pointed by q**

Now, combining the meaning of both lvalue and rvalue, we conclude that:

*&a **value at address (&a) pointed by p**	= is set to	*p **value at address pointed by q**	+ +	*q **value at address**

OR,

value at address (&a) is set to (addition of **value at address pointed by p and value at address pointed by q).**

Thus, a =10 + 10 = 20 /* In line 3, *p =10 and q also refer to address of a */

Line 5: a = *p * *q;

Here * just after p denotes multiplication symbol. Thus **a** is set to multiplication of **value at address pointed by p and value at address pointed by q**. Previously at line 4, a becomes 20. So after executing line 5, a = 20 * 20 = 400.

Look at program 1.5 and execute it on your machine. Also verify the results.

```
/* Program 1.5   */
#include <stdio.h>
int main()
{               int a = 8 , b = 4, c, *p, *q, *r;
                p = &b;
                q = p;
                r = &c;
                q = &a;
                *q = 10;
                *r = *p+7;
                *r = a + *q + *&c;
                printf("%d %d %d\n", a, b, c);
                printf("%d %d %d\n", *p, *q, *r);
                return 0;
}
```

The output of the following program is:

```
10  4   31
10  4   31
```

The following Program 1.6 performs addition of two numbers using pointers. Create the program in your machine, execute it and verify the results.

```
/* Program 1.6   */
#include <stdio.h>
int main()
{
                int x , y, sum;
                int *px = &x;
                int *py = &y;
                int *pr = &sum;
                printf("Enter the first number    : ");
                scanf("%d", px);
                printf("Enter the second number : ");
```

```
        scanf("%d", py),
        *sum = *x + *y;
        printf("\n%d + %d is %d", *px, *py, *sum);
        return 0;
    }
```

The output of the program is:

```
Enter the first number    : 34
Enter the second number : 56
34 + 56 is 90
```

1.5 POINTER ARITHMETIC

In section 1.3 we have already explained that 'C' pointer is an address which has a numeric value. As a result, you can perform arithmetic operations on a pointer. There are four arithmetic operators that can be used on pointers:

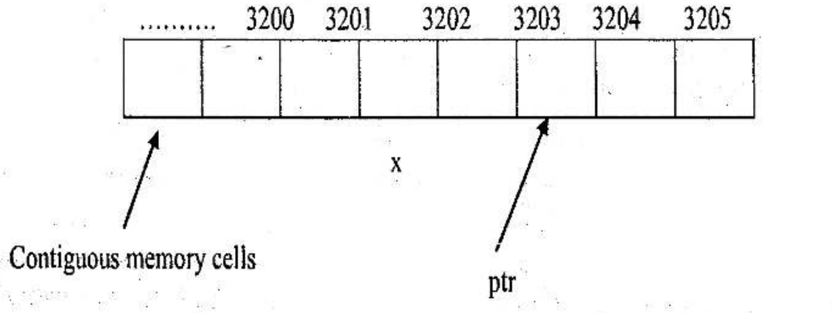
- Integers and pointers can be added and subtracted from each other, i.e. + and -
- incremented and decremented, (++ and --)
- In addition, different pointers can be assigned to each other

Let us assume,that **ptr** is an integer pointer which points to the address 3202. Now, perform the following arithmetic operation on the pointer:

ptr = ptr + 1;

Now after executing the above statement, the **ptr** will point to the **nextvalue** of its **type**., i.e. 3204. This is because when we increment a pointer, its value is **increased** by the **length** of the **datatype** that it points to. This operation will move the pointer to next memory location without effecting actual value at the memory location. If **ptr** points to a character whose address is 1200, then above operation will point to the location 1201 because next character will be available at 1201.

[Note] A variable's address is the first byte occupied by the variable.Therefore whenever pointer points to a variable, it's actually pointing to the first byte of address. See Figure 1.4 for illustration.**[End of Note]**



(a) After the statement, ptr = ptr +1, ptr jump **2** bytes as **int** takes 2 bytes in memory

Figure 1.4: illustration of pointer movenment

1.5.1 Pointer Increment

It is more easy to increment a pointer variable as compared to array. Array name cannot be incremented because it is a constant pointer. The following program 1.7 increments the pointer variable to access each successive element of the array:

```
/*  Program 1.7   */
#include <stdio.h>
const int MAX = 3;
int main ()
{         int arr[] = {100, 50, 120};
          int i, *ptr;
          /* let us have array address in pointer */
          ptr = arr; /* name of array is actually base address of
                        array   */
          for ( i = 0; i < MAX; i++)
{
          printf("Address of arr[%d] = %x\n", i, ptr );
          printf("Value of arr[%d] = %d\n", i, *ptr );
          /* move to the next location */
ptr++;
}
return 0;
}
```

The output of the above program is:

```
Address of arr[0] = 3205
Value of arr[0] = 100
Address of arr[1] = 3207
Value of arr[1] = 50
Address of arr[2] = 3209
Value of arr[2] = 120
```

1.5.2 Pointer Decrement

The same concept also applies to decrement the pointer, which decreases its value by the number of bytes of its data type as shown in Program 1.8:

```
/* Program 1.8 */
#include <stdio.h>
const int MAX = 3;
int main ()
{
int arr[] = {100, 50, 120};
int i, *ptr;
/* let us have array address in pointer */
ptr = &arr[MAX-1];          /* pointer contains address of last
element */
for ( i = MAX; i > 0; i—)
{
printf("Address of arr[%d] = %x\n", i, ptr );
printf("Value of arr[%d] = %d\n", i, *ptr );
/* move to the previous location */
ptr—;
}
return 0;
}
```

The output of the above program is:

```
Address of arr[3] = 3209
Value of arr[3] = 120
Address of arr[2] = 3207
Value of arr[2] = 50
Address of arr[1] = 3205
Value of arr[1] = 100
```

1.5.3 Comparing pointers

You can also compare pointers by using relational operators, such as ==, <,<=, > and >=. If ptr1 and ptr2 point to variables that are related to each other, such as elements of the same array, then p1 and p2 can be compared.

The following program 1.9 modifies the program 1.7 by incrementing the variable pointer in such a way that the address to which it points is either less than or equal to the address of the last element of the array, which is &arr[MAX - 1]:

```
/*  Program 1.9  */
#include <stdio.h>
const int MAX = 3;
int main ()
{
int arr[] = {100, 50, 120};
int i, *ptr;
ptr = arr;      /* assign the address of arr to pointer variable */
i = 0;
while ( ptr <= &arr[MAX - 1] )
{
printf("Address of arr[%d] = %x\n", i, ptr );
printf("Value of arr[%d] = %d\n", i, *ptr ); /* point to the previous location
*/
ptr++;
i++;
}
return 0;
```

}

The output of the above program is:

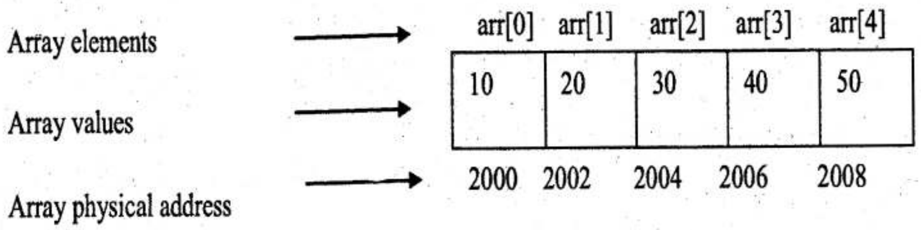
```
Address of arr[0] = bfdbcb20

Value of arr[0] = 100

Address of var[1] = bfdbcb24

Value of arr[1] = 50

Address of var[2] = bfdbcb28

Value of arr[2] = 120
```

1.6 ARRAYS AND POINTERS

As we know that, when an array is declared, the compiler allocates a **baseaddress** and sufficient amount of memory to store all the elements of an array in contiguous memory locations. The **base address** refers to **index0** or **firstelement** of an array. Also, **name of an array** is defined as **constant pointer** to the first element, so it **cannot** be used as **lvalue.** Consider the following statement:

static int arr[5] = { 10, 20, 30, 40, 50};

Assume that its base address is 2000 and assuming 16 bit machine, int takes 2 bytes in memory. The five elements stored in memory is shown below. Every next element is 2 bytes apart from each other.



Suppose we point our integer pointer **ptr** to this array **arr**. The statement looks like;

int *ptr;

ptr = &arr[0];/* Line 1: assigning base address to integer ointer ptr */

Since, name of an array **itself** represents base address, this implies hat

ptr = arr; /* Line 2: assigning base address to integer pointer tr */

ssigns base address of array **arr** to integer pointer **ptr**. Thus, both the bove statements (Line 1 and Line 2) are same.

Using indirection operator, we can say that *ptr is same as arr[0].

Now, what happens when we write:

ptr + 1;

Since the compiler "**knows**" this is a pointer (i.e. its value is an address) and that itpoints to an **integer** (its current address, 2000, i.e. address of an integer), it adds 2 to **ptr** instead of 1, so the pointer "points to" the **next integer**, at memory location 2002.Similarly, the **ptr** declared as a pointer to a long, it would add 4 to it instead of 1.The same goes for other data types such as floats, doubles, or even user defined datatypes such as structures.

Returning to our discussion, ptr +1 points to memory location 2002, which is the address of arr[1]. Again applying indirection operator, we can say that *(p+1) is same as arr[1]. Applying the same process repeatedly, we observe that:

ptr	= &arr[0]	(= 2000)	*ptr	= arr[0] = 10
ptr +1	= &arr[1]	(= 2002)	*(ptr + 1)	= arr[1] = 20
ptr +2	= &arr[2]	(= 2004)	*(ptr + 2)	= arr[2] = 30
ptr +3	= &arr[3]	(= 2006)	*(ptr + 3)	= arr[3] = 40
ptr +4	= &arr[4]	(= 2008)	*(ptr + 4)	= arr[4] = 50

Therefore, in general, if we increment the pointer **ptr** by **i** increment step, the point moves to next location of its type. Thus,

***(ptr + i) = arr[i] = *(arr + i)** / since ptr = arr; refer Line 2 */

The address of an element is calculated using its index and sizeof(data_type)as:

Address of arr[2] = base address + (2 * sizeof(int)) = 2000 + 2*2 = 2004.

Using pointers in this way leads to very efficient code and faster accessing than array indexing.

Consider the following three programsegment that shows how to sum up all the elements of a 1D array usingpointers. Their implementations are completely different.

Method 1: The elements are summed up one by one and addition is stored in variable sum

```
int a[100],i,*p,sum=0;
for(i=0; i<100; ++i)
sum +=a[i];
```

Method 2: The elements are summed up using indirection operator applied on array **a**

```
int a[100], i, *p, sum = 0;
for(i = 0; i < 100; ++i )
sum += *(a + i );
```

Method 3 The elements are summed up using indirection operator applied on **p**

```
int a[100],i,*p,sum=0;
for(p=a; p<&a[100]; ++p)
sum += *p;
```

Let us consider another example where the following code shows three ways in which an array can be accessed. The program computes square of position of element.

```
int square[5] = { 10, 20,, 30, 40, 50};
int *ptr = &square[0];
 /* assigns square of number to element */
for (i = 0; i < 5; i++)
square[i] = (i+1)*(i+1);
/* and so does this */
for (i = 0; i < 5; i++)
 ptr[i] = (i+1)*(i+1);
/*and this - note comma*/
for(i = 0;i < 5;ptr++, i++)
*ptr = (i+1)*(i+1);
```

Next program 1.10 uses **sizeof**() operator to determine how much bytes are taken by the data types to which pointer referring and also the size of pointers on the machine. Execute the program in your machine and verify that how much bytes are taken by pointers.

```
/* Program 1.10  */
int main()
{
        char Ch;
        char *PC;
        int sizeofCh = sizeof(Ch);
        int sizeofPC = sizeof(PC);
        int sizeofStarPC = sizeof(*PC);
        int A;
        int *PA;
        int sizeofA = sizeof(A);
        int sizeofPA = sizeof(PA);
        int sizeofStarPA = sizeof(*PA);
        double D;
        int *PD;
        int sizeofD = sizeof(D);
        int sizeofPD = sizeof(PD);
        int sizeofStarPD = sizeof(*PD);
        printf("sizeof(Ch) : %3d  | ", sizeofCh);
        printf("sizeof(PC) : %3d  | ", sizeofPC);
        printf("sizeof(*PC) : %3d  | ", sizeofStarPC);
        printf("sizeof(A) : %3d  | ", sizeofA);
        printf("sizeof(PA) : %3d  | ", sizeofPA);
        printf("sizeof(*PA) : %3d  | ", sizeofStarPA);
        printf("sizeof(D) : %3d  | ", sizeofD);
        printf("sizeof(PD) : %3d  | ", sizeofPD);
        printf("sizeof(*PD) : %3d  | ", sizeofStarPD);
        return 0;
}
```

The output of the above program is:

```
sizeof(Ch) :   1  |  sizeof(PC) :   2  |  sizeof(*PC) :  1
sizeof(A) :    2  |  sizeof(PA) :   2  |  sizeof(*PA) :  2
sizeof(D) :    8  |  sizeof(PD) :   2  |  sizeof(*PD) :  8
```

Pointers can also be used to perform operations on two-dimensional arrays. In a one-dimensional array **x**, the expression

***(x + i) = *(p + i) = x[i]** are same.

Likewise, an element in a two dimensional array can be represented by the pointer expression as follows:

***(*(x + i) + j) = *(*(p + i))** which is same as **x [i][j]**

1.7 CHARACTER STRINGS AND POINTERS

A string is a variable-length array of characters that is delimited by the null character. A string literal or string constant is enclosed in double quotes. You will surprise to know that a string constant like "Hello World"is treated by the compiler as an address (Just like we saw with an array name).The value of the string constant address is the **base address** of the characterarray. Consider the statements:

```
char *str = "Hello World";   /* Line 1  */
char *ptr;                   /* Line 2  */
ptr = str;                   /* Line 3  */
```

In line 1: str is a pointer to character and assigns the address of the string constant "Hello world". This implies that address of first element 'H' is assigned to str;

In line 2, ptr is a pointer to character.

In line 3, address stored in str is assigned to pointer variable ptr.

So, & str is same as str.

Thus, we can **use pointers to work with character strings**, in a similar way that we used pointers to work with "standard" arrays. This is demonstrated in the following code:

```
#include <stdio.h>
void main() {
        char *cp;
        cp="Hi Dear";
        printf("%c\n",*cp);
        printf("%c\n",*(cp+6));
}
```

The output of the above program is:

```
H
r
```

[Note] The assignment made in Line 1 above is not applied to character arrays. The statement like

```
char str[30];
str = "Hi Dear";
```

do not work.**[End of Note]**

Let us consider, another example which illustrates easy string input using pointers:

```
#include <stdio.h>
main() {
        char *msg;
        printf("How are you?\n");
        scanf("%s",msg);
        printf("Hi Dear! %s\n", msg);
}
```

The output of the above program is:

```
How are you?
Hi Dear! I am fine.
```

[If **I am fine** is entered from the keyboard during the scanf operation]

Consider the program 1.11 which is our own version of computing length of a string.

```
/* Program 1.11 */
#include <stdio.h>
void main() {
        char *msg;
        char *ptr;
        int len=0;
        printf("Enter a string: ");
        scanf("%s", msg);
        ptr = msg;
        while(*ptr != '\0')
        {
                len++;
                ptr++;
        }
        printf("Length of %s is %d\n", msg, len);
}
```

The output of the program is:

```
Enter a string: Hello
Length of Hello is 5.
```

The program 1.11 can be modified in a more compact way. See program 1.12.

```
/* Program 1.12 */
#include <stdio.h>
void main() {
        char *msg, *ptr;
        int len=0;
        printf("Enter a string: ");
        scanf("%s", msg);
        ptr = msg;
        while(*ptr++) len++;
        printf("Length of %s is %d\n", msg, len);
}
```

Another way to compute length of a string is shown in Program 1.13.

```
/* Program 1.13 */
#include <stdio.h>
void main() {
        char *msg, *ptr;
        printf("Enter a string: ");
        scanf("%s", msg);
        for(len =0, ptr = msg; *ptr ; len++, ptr++);    /* Note: semi-colon
here */
        printf("Length of %s is %d\n", msg, len);
}
```

Question 1: Execute the program 1.11 to program 1.13 and verify the results.

1.8 ARRAY OF POINTERS

There are many situations when we need to maintain an array which can store pointers to an **int** or **char** or any **other** data type available. The declaration of an array of pointers to an integer variable is given below:

int *ptr[MAX];

The above statement declares **ptr** as an array of **MAX** integer pointers. Thus, each element in **ptr**, now stores a pointer to an **int** value. The following example uses three integers which will be stored in an array of pointers as follows:

```
#include <stdio.h>
const int MAX = 3;
int main ()
{
        int arr[] = {100, 50, 120};
        int i, *ptr[MAX];
        for ( i = 0; i < MAX; i++)
        {       ptr[i] = &arr[i];          /* assign the address of integer. */
        }
        for ( i = 0; i < MAX; i++)
        {
                printf("Value of arr[%d] = %d\n", i, *ptr[i] );
        }
        return 0;
        }
```

The output of the above program is:

```
Value of arr[0] = 100
Value of arr[1] = 50
Value of arr[2] = 120
```

The next example uses an array of pointers to character to store a list of strings as follows:

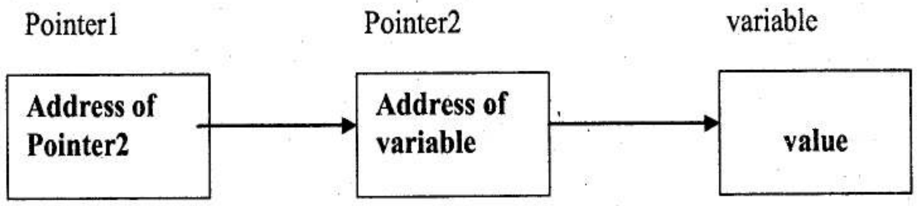
```
#include <stdio.h>
const int MAX = 4;
int main ()
{
        char *names[] = {
        "Sammer",
        "Himanshu",
        "Nishant",
        "Suraj",
};
        int i = 0;
        for ( i = 0; i < MAX; i++)
        {
                printf("Value of names[%d] = %s\n", i, names[i] );
        }
        return 0;
}
```

The output of the above program is:

```
Value of names[0] = Sammer
Value of names[1] = Himanshu
Value of names[2] = Nishant
Value of names[3] = Suraj
```

1.9 POINTER TO POINTER

A pointer to a pointer is a type of **multiple indirection**, or a chain of pointers. Usually, a pointer contains the address of a variable. Now, we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below:



A variable that is pointer to a pointer must be declared as such. This can be done by putting an additional asterisk in front of its name. For example, following is the way to declare a pointer to a pointer of type int:

int **var;

When value of a variable is indirectly pointed by a pointer to a pointer, accessing that value requires that the asterisk operator be applied **twice**, as is shown below in the example:

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

```
        int *ptr;
        int **pptr;
        var = 230;
        /* take the address of x */
        ptr = &x;
        /* take the address of ptr using address of operator & */
        pptr = &ptr;
        /* take the value using pptr */
        printf("Value of x= %d\n", x);
        printf("Value available at *ptr = %d\n", *ptr );
        printf("Value available at **pptr = %d\n", **pptr);
        return 0;
}
```

The output of the above program is:

```
Value of x= 230
Value available at *ptr = 230
Value available at **pptr = 230
```

Check your Progress 1

1. Consider the following declaration

int a, *b = &a, **c = &b;

What following program fragment does?

```
a = 4;
**c = 5;
```

2. What will be the output of the following code segment?

```
void main( )
{   char s[10];
        strcpy(s, "abc");
        printf("%d %d",
         strlen(s), sizeof(s));
}
```

3. What is the output of the following 'C' program?

```
# include <stdio.h>
void main ( ){   int a, b=0;
        static int c [10]={1,2,3,4,5,6,7,8,9,0};
        for (a=0; a<10;+ + a)
         if ((c[a]%2)= = 0) b+ = c [a];
        printf ("%d", b);
}
```

4. What is the output of following program:-

```
int q, *p, n;
q = 176; If the address of q is 2801
p = &q; and p is 2600
n = *p;
printf("%d", n);
```

5. The size of array int a[5]={1,2} is ____________

6. The output of the following statements is

```
        char ch[6]={'e', 'n', 'd', '\0', 'p'};
        printf("%s", ch);
```

7. Given the following code fragment:

```
        int main()
{
        int raw[20], i, sum=0;
        int *p = raw;
        for (i=0; i < 20; i++)
                *(p+i) = I;
        for(i=0; i < 20; I += sizeof(int))
        sum += *(p+i)
        printf("sum = %d \n", sum);
        return();
        }
```

What will be the result of execution?

8. What is the missing statement in the following function which copies string x into string y.

```
void strcpy( char *x, char *y)
{
while (*y != '\0')..................... /* missing stament */
*x = '\0';
}
```

9. Write a 'C' program to calculate the frequencies of different alphabets, present in a given string. The string of alphabets is to be taken as input from the keyboard.

10. Assuming the following declarations:

```
        int *pi;int i, j;
        char *pc;
```

```
char c[ ] = "abracadabra";
```

what are the final values of i, j, and c after executing the following statements.

```
i = 1;
j = 2;
pi = &i;
*pi = 3;
pc  = 3;
pc += 3;
*pc = 'R';
—j;
```

11. What thing(s) is/are wrong with the following code fragment? [Note: there may be one thing wrong].

```
int array[] = {1, 2, 3};
int i, sum = 0;
for (i = 0; i <= 3; i++) {
        sum += array[i];
}
```

12. Under what conditions can the following fragment lead to program failure?

```
char input[80];
scanf("%s", input);
```

13. What is wrong with the following code?

```
{char *s1 = "Hello, ";
char *s2 = "world!";
char *s3 = strcat(s1, s2);
}
```

14. What happens with this code? Please explain

```
char* s = "hello\0";
char buffer[20];
printf("%s is of length %d\n",s, strlen(s));
strcpy(s, buffer);
```

15. What is the output of the following program:

```
char name[30] = "Allahabad";
```

```
/* more code here that might change contents of name */int i = 0;
while ( i < 30 && name[i] != '\0' ) {name[i] = toupper( name[i] );
i++;}
```

1.10 SUMMARY

In this Unit, we have studied pointers and their relationship to arrays and character strings. Pointers are very useful part of 'C'and separate it from more conventional programming languages. Pointers make 'C' more influential allowing a wide variety of tasks to be accomplished. A pointer is a constant or variable that contains an address which can be used to access data. Pointers are built on the fundamental concept of pointer constants. The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer referring to. The unary operator *, also known as indirection (or dereferencing) operator returns the value of the variable located at the address specified by its operand. The indirection operator is complement to the address operator. The name of an array is defined as constant pointer to the first element, so it cannot be used as lvalue. Similarly, we can use pointers to work with character strings, in a similar way that we used pointers to work with "standard" arrays.

UNIT 2: FUCNTIONS

Structure

2.0 INTRODUCTION

The programs we have seen so far werevery simple. They simply solved the given problem that may be carried out without too much effort. The principles of top–down design and structured programming state that a program should be divided into a main module and its related modules. Each module is then further divided into sub-modules (or pieces)until the resulting modules are basic; that is, they cannot be divided into further sub-sub-modules.These smaller pieces sometimes called **'modules'** or **'subroutines'** or **'procedures'** or **functions** (see Figure 2.1). Visualize that you have to develop a huge program like operating system or word processor, which includes large number of coding lines. It is not feasible toimplement the whole program in a one big program file. It is the work of many computer programmers, each working on smaller pieces of the problem that are then brought together to complete the solution. Thus, without using functions, the program becomes very clumsy, hectic and complex to understand. After designing the program, one has to check that whether it is functioning well or not. This can be done only when the big program is divided into smaller functioning units. So, the needs of functions are:

- It helps to manage complexity because smaller blocks of code are easily readable and understandable.
- You can re-use the code, within a particular program or across different programs.
- It also allows independent development of code.

- Functions provide a layer of 'abstraction'. Abstraction is a way to simplify or separate the details of how a process works to an essential set of features that allow the process to be used. For example: **pritnf()** is used to display the content on to the monitor. You need not worry abouthow it works, rather you simply use it in your program.

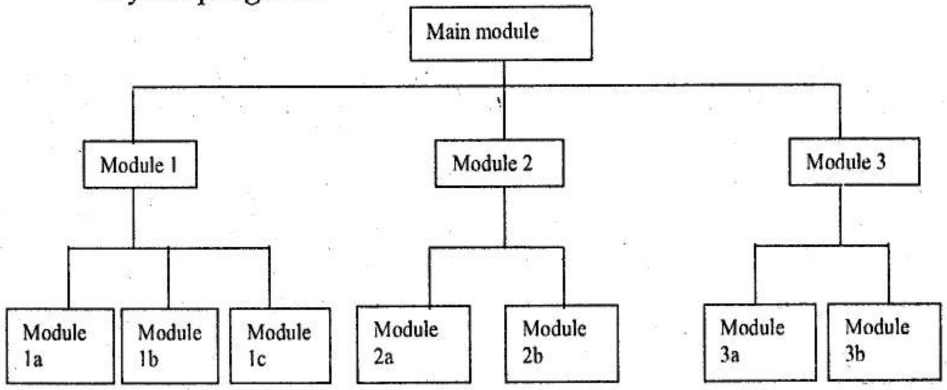


Figure 2.1: Top-down approach using modules (or functions)

2.1 OBJECTIVES

After going through this unit, you should be able to:

- declare function prototypes
- create user defined functions to perform task
- call functions by value
- call functions by reference
- pass arrays as arguments to functions

2.2 FUNCTIONS

In 'C', the initiative of top–down design is done with the help of functions. A 'C' program is made of one or more functions, one and only one of which must be named **main**. The execution of the program always starts with main, but it can call other functions, including library function such as **printf()** and **scanf()**, to do some part of the task. A task is a distinct job that your program must perform as a part of its overall operation, such as adding two or more integer, sorting an array into numerical order, or calculating a square root etc. Generally, the purpose of a function is to receive zero or more pieces of data, operate on them, and return at most one piece of data. A function will carry out its intended action whenever it is accessed (i.e., whenever that function is "**called**") from some other portion of the program. The same function can be accessed from several different places withi a program. Once the function has carried out its intended action, control will be returned to the point from which the function was accessed as shown in Figure 2.2.

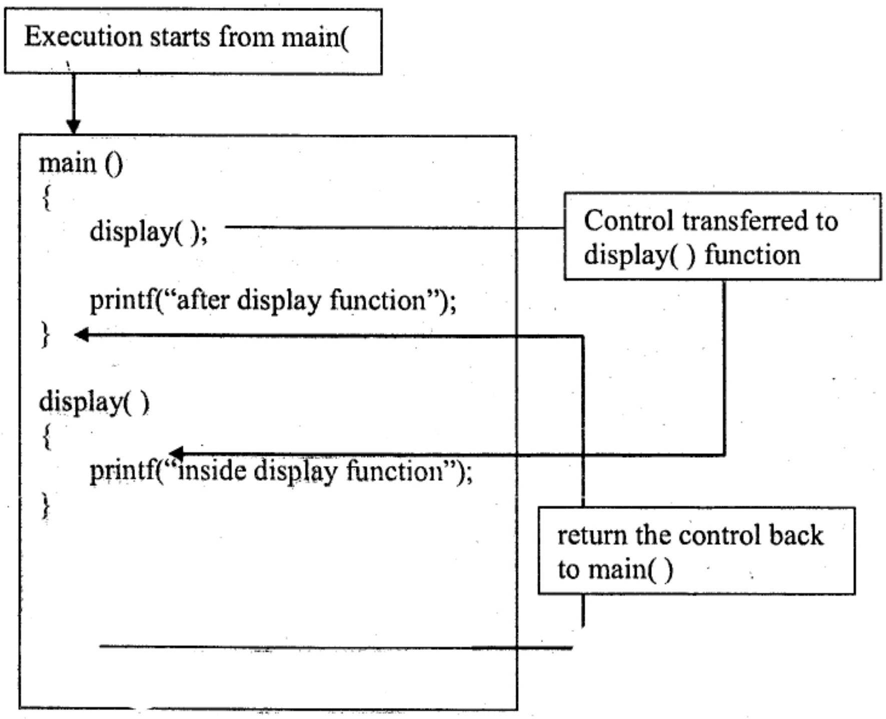

Execution starts from main(
main ()
{
display( );
printf("after display function");
}
display( )
{
printf("inside display function");
}
Control transferred to display( ) function
return the control back to main( )

Figure 2.2: calling and defining function

Function can be classified into two categories: *library functions* and *user defined functions*. We have seen number of **library** functions in previous Units, like; **printf, scanf, strlen, getchar, putchar** etc. These functions are defined in standard C library files. In this Unit, we focus on user defined functions that are designed for our intended task.

User defined function names follow the **same rule** as **variablenames**. Functionnames consist of **letters, numbers** and**underscores**but **do not** start their name with number. Let's take an example. Suppose you need a function which returns the larger of two integers. The function will define as:

```
int maximum (int x, int y)

/* Return the largest integer */

{

  if (x>y) return x;

  return y;

}
```

The above function *maximum* takes two *arguments* both of which are integers. It returns the value of the largest integer. Note that **return** is being used here in the same way as **return** was used from **main** but returning from main exits the program whereas returning from a function transfers control to wherever the function was called from. Now, let us use our maximum function from *main* function to compute the maximum among two integer numbers as specified in Program 2.1

```
/* Program 2.1   */
#include <stdio.h>
int maximum (int, int);    /* Note :  Prototype or declaration, we
                                      will see it later */
int main()
{
   int i= 4;
   int j= 5;
   int k;
   k= maximum ( i ,j);              /* Calling maximum function */
  printf ("%d is the largest among %d and %d\n",k,i,j);
   return 0;
}
int maximum (int a, int b)         /*   function   definition   or
implementation of function */
/* return the largest integer */
{
 if (a > b) return a;
   return b;
}
```

In the above program, we are making our own function *maximum* that computes the maximum among two integer numbers input to it. The program has three features, namely; **prototype** or **declaration**, function **definition** and function **calling**. A function **declaration** tells the compiler about a function's *name, returntype,* and *parameters*. A function **definition** provides the *actual body* of the function. The function calling specifes the place in main from where the function is used. In next section we will describe them in more detail.

2.3 USER-DEFINED FUNCTIONS

Like every other variable in 'C', functions must be both declared and defined. The function declaration gives the whole representation of the function that needs to be defined later. The function definition contains the actual code for a function.In order to use functions, the programmer must do three things:

- **Define** the function
- **Declare** the function
- **Calling** the function in the main code.

2.3.1 Defining a Function

The general form of a **function definition** is:

```
return_typefunction_name(parameter list)/* function header */
{
            body of the function          /* function body */
}
```

A function **definition** consists of a **functionheader** and a **functionbody**.

(a) The function **header** has following parts:

- **Return Type**: A function may return a value. The **return_type** is the data type of the value the function returns and it may be any valid 'C' data type, including user defined data type. There are some functions that perform the desired operations without returning a value. In this case, the **return_type** is the keyword **void**. The term void denotes nothing returned by the function.

- **Function Name:** This is the actual name of the function. Function names follow the **same** rule as **variable names**. Functionnames consist of **letters, numbers** and **underscores** but **do not** start their name with number. The function **name** and the **parameter list** together constitute the **function signature**.

- **Parameters:** When a function is **called** (or **invoked**) from calling program, you pass a value to the parameter. This **value** is referred to as **actual parameter** or **argument**. The parameter list refers to the **type**, **order**, and **number** of the parameters of a function The parameter list may be **optional,** i.e., a function may contain no parameters. **Formal parameters** are variables that are declared in the header of the function definition.

(b) **Function Body:** The function body actually contains the collection of compound and simple statements that define what the function does. For example in program 2.1, the function maximum computes the maximum among two integer numbers

For example, if we define a function to return the square of a number, then function definition looks like:

```
/* function to compute squared of a number   */
int squared (int i)       /* Note:   Here i is a formal parameter   */
/* Squares i */
{
  i= i*i;
  return i;
}
```

The above function header consists of following three parts, **return type** of function is **int**, **name** of function is **squared**. The **parameter list** contains **int** type value. The first **int** tells the compiler that the value returned by the function will beconverted, if necessary, to int.The parameter tells the compiler that the function takes a single argument of type**int**.The body of function simply squared the value and returns it to calling function.Any variables declared in the body of function are said to be **local** to that function. In our example, variable ***i*** is local as it is declared inside the function maximum. Other variables may be declared external to the function. These are called **global**variables.

By using the keyword **return** followed by a data variable or constant value, a function returns a value to the calling program. The **return** statement may also contain an expression. Some examples of return statements are:

```
return 5;
return  (( n + 1) * ( n + 1));
return a*b*c;
```

When a **return** keyword is encountered the following actions are performed:

- execution of the function is finished and control is transferred back to the calling program,
- The function call evaluates value of the ***return*** *expression*.

But if there is no **return** statement control is simply transferred back when the closing brace of the function is encountered.

The data type of the *return expression* **must match** with the declared **return_type** for the function.

It is also possible that a function have multiple **return** statements. For example:

```
double absolute(double x) {
if (x>=0.0)
return x;/* if received value x is greater than 0, then simply return
it. */
else
return -x;      /* otherwise, negative of x is returned   */
}
```

There are many functions thoseactually neither return any value nor they require any arguments so for these functions the keyword **void** is used. Here is an example:

```
void write_header(void)
{
        printf("Hi! This is the print message\n ");
        printf("Welcome to the world of C programming\n");
        printf("You are about to learn fucntions ");
}
```

The 1st **void** keyword before function name indicates that **no value** will be returned.The 2nd **void** keyword in parameter list indicates that **no arguments** are needed for the function.This makes logic because all this function does is print out a header statement.

[Note]In 'C', If a function definition does not specify the return type, then it is **int** by default.**[End of Note]**

2.3.2 Declaring a function

Like any other 'C' variable, functions should be declared before they are used. ANSI 'C' provides for a new function declaration syntax called the **function prototypes.** A**function declaration** tells the compiler about a function **name**, the **number** and **types** of arguments that are passed to the function and the **type** of value that is to be returned by the function. The actual **body of the function** can be defined separately, which we have seen in previous section. A general form of function **declaration**is shown below:

return_type function_name(parameter type list);

- The parameter type is a typically a comma separated **list of types.** Identifiers areoptional; they do not affect the function prototype.
- The keyword **void** is used if a function takes no argument.
- Alsothe keyword **void** is used ifno value is returned by the function.

For the above defined function **squared()**, following is the function declaration:

int squared (int i);

Parameter names are not important in function declaration only their type is required, so following is also valid declaration:

```
int squared(int);            /* also a valid declaration */
```

Function declaration is required when you define a function in one source file and you need to call that function in another file. In such a case, you should declare the function at the top of the file calling the function.

The above function prototype of squared is simply the **function header** from the function definition **with a semi-colon attached to the end**. The prototype tells the compiler about the **number** and **type** of the arguments to the function and the **type** of the return value. Function prototypes should be placed **before the start of the main program**. The function definitions can then follow the main program. In addition to make code more readable, the use of function prototypes offers **improved type checking** between actual and dummy arguments.

2.3.3 Calling the function

Calling a function is very easy. A function which calls another function is known as **calling function** whereas function to be called is known as **called function**. To call a function, **just type its name** in your program and make sure to supply arguments (if required by your function prototype). A statement to call our function **squared(int i)**, we simply write:

```
squr = squared(5);
```

where variable **squr** is defined in a calling function.

When your program encounter the function calling, control passes to the called function. If the task of function is completed, control passes back to the **main**(or **calling**) program. Additionally, if a value was returned, the function call takes on that return value. In the above example, upon return from the **squared** function the statement:

```
squared(5);
```

returns **25** and that returned integer value is assigned to the variable **squr**.

Therefore on combining function definition, function prototype and function calling, we finally build our first user defined function for squaring an integer number. The following program 2.2 uses function **squared()** to compute square of an integer which is passed to it.

```
/* Program 2.2 */
#include<stdio.h>
int squared (int i );          /* Prototype or declaration */
int main()
{
  int i= 4;
  int squr;
squr= squared ( i );                /* Calling function */
  printf ("The square of %d is &d\n", i, squr );
  return 0;
}
int squared (int i)          /*    Function definition. The i is
theformalparameter    */
{
i = i*i;
  return i;
}
```

Actual parameters are variables or constants thoseare passed to the **called function**. In this example, value **4** is an actual parameter. Formal and actual parameters must match exactly in type, order, and number. Their names, however, do not need to match.

There are following important tips which should be kept in the mind before calling function.

- The number of arguments in the function call must match the number of arguments in the function definition. The number of actual and formal arguments should same.
- The type of the arguments in the function call must match the type of the arguments in the function definition.
- The actual arguments in the function call are matched in-order with the formal arguments in the function definition.
- The actual arguments are passed by-value to the function. The formal parameters in the function are initialized with the present values of the actual arguments.

2.4 CATEGORIES OF FUNCTION

A function can be categorized according to the number of parameterspresent and return type (i.e. no return value or some return value). The functions can belong to any one of the following categories:

1. void Functions without parameters
2. *void* Function with Parameters
3. Non-void Function without Parameters
4. Non-void Functions with Parameters

2.4.1 void Functions without parameters

This type of function has return type **void** which indicates that the function does not return anything back to the calling function. The function also does **not** receive any data from the calling function. Thus there is no data transfer among calling and called function. You can say that there is only silent communication takes place between the functions **without** any **give** and **take** action. The following program 2.3 requires a positive integer as input from user. When user enters any negative number than warn_and_quit() function invoked. The function displays message and program is abnormally terminate by **exit(1)** function. This is a standard library function that causes termination of program. Conventionally, a return value of 0 signals that all is well; non-zero values usually signal abnormal situations.

```
/*  Program 2.3   */
#include<stdio.h>
/* function warn_and_quit has void return type. Also, its parameter list
is empty(void ) */
void warn_and-quit(void );            /* Prototype or declaration */
int main()
{
   int x;
  printf("Enter only positive number, Otherwise you suffer !: ");
   do { scanf("%d", &x);
        if( x < 0 )
        warn_and_quit(  );            /* calling function */
        } while ( x < 0 );
   return 0;
}
 /*  function definition  */
 void warn_and_quit ( void)        /* Here i is a formal parameter   */
 {
        printf("You have to enter only positive number\n');
        printf("We have given you warning, but you neglet it. So I am
        quit. \n");
   exit(1);       /* abnormally terminate the program at this end */
 }
```

2.4.2 *void* Function with Parameters

In this category, an argument should be passed from the calling function but the called function won't return anything back to the calling function. The calling function sends the data to the called function but did not receive anything from the called function. This is a kind of **downward** (one way)communication from calling to called function. Program 2.4 checks that whether a number input by user is even or odd.

```
/* Program 2.4 */
#include<stdio.h>
/* function check_even_odd(int ) has void return type. It has one int
parameter */
void check_even_odd (int number );                                    /     *
Prototype or declaration */
int main()
{
        int num;
        printf("Enter only positive number: ");
        scanf("%d", &num);
        while ( num<= 0 )
{
        printf ("\nThat's incorrect. Try again.\n") ;
        printf ("Enter a positive integer: ") ;
        scanf ("%d", &num) ;
}
check_even_odd(num );      /* Note: num is actual argument   */
  return 0;
}
/* function definition */
void check_even_odd( int number)      /* Here number is a formal
parameter   */
{
        If( number % 2 == 0)
                printf("The number %d is even\n", number);
        else
```

```
            printf("The number %d is odd\n", number);
}
```

In the above program, an integer number is requested form user. If user types incorrect, than again he is requested to entered the positive number. This process will continue till the correct number is not entered. On receiving correct number, the function **check_even_odd** is called with actual argument **num**. This argument **num** is passed to called function **check_even_odd**. The number **num** is received by **formal parameter number**. The functionchecks whether **number % 2** gives remainder 0 or not. If remainder is zero than number is **even** otherwise it is **odd**. Subsequently corresponding **printf** statement is executed.

You must sure that **formal** parameters and **actual** parameters must match in **number, type** and **order**. The values of actual parameters are assigned to the formal parameters on a one-to-one basis, starting from left to right.

2.4.3 Non-void function without parameters

The function of this category **returns some value** to the calling function but **no arguments are passed** from the calling function to the called function. The calling function simply invoked the called function for some computation. The called function performs some operations and sends the data back to the calling function. It is a kind of **upward**(one way) communication as data is transferred from called to calling function. Program 2.5 checks that whether a number input by user is even or odd.

```
/*  Program 2.5   */
#include<stdio.h>
/* function int check_even_odd(void) has int return type. It has no
parameter, i.e. void */
intcheck_even_odd (void);           /* Prototype or declaration */
int main()
{
  int num, flag = 0;
  flag = check_even_odd( ); /*  Note: flag collects the value returned
by the function  */
   if( flag ==1 )
        printf(" Number is even");
   else
        printf(" Number is odd");
```

```
    return 0;
}
/* function definition */
intcheck_even_odd(void)                    /*    Here number is a
formalparameter    */
{       printf("Enter only positive number: ");
        scanf("%d", &num);
        while ( num<=0 )
{
printf ("\nThat's incorrect. Try again.\n") ;
printf ("Enter a positive integer: ") ;
scanf ("%d", &num) ;
}
        if( number % 2 == 0)
        return 1;       /* if num is even , return the value 1 */
        else
        return 0;       /* if num is odd , return the value 0 */
}
```

The above function maintains a **flag** variable in main( ). The called function requires an input from user. If user types even number than return value is 1 otherwise 0 is returned to the calling function. This returned value is assigned to flag. The flag value is then checked against 1 or 0. If return value is 1, the number is even otherwise it is odd. This program is another version of program 2.4.

2.4.4 Non-void Functions with Parameters

In this category of functions, the calling function send some data to called function, which in turn perform some operations over that data and sends back the computed result to the calling function. The function has **both return type** as well as it has **formal parameter** list. It is a kind of two way (bi-directional) communication because both functions interact with each other by sending data to one another. Program 2.6 computes the multiplication of two integer numbers by sending them to the called function.

```
/* Program 2.6 */
#include<stdio.h>
/* function int multiply(int a, int b) has int return type. It has two int
parameters */
```

```
intmultiply (int a, int b );     Prototype or declaration */
int main()
{
        int num1, num2, result;
        printf("Enter two positive integers: ");
        scanf("%d %d", &num1, &num2);
        while ( (num1<= 0) | | ( num2 < =0))
{
        printf ("\nThat's incorrect. Try again.\n") ;
        printf ("Enter two positive integers: ") ;
        scanf("%d %d", &num1, &num2);
}
        result = multiply(num1, num2);
        printf("Multiplication of %d and %d is %d\n", num1, num2,
result);
        return 0;
}
/*  function definition  */
int multiply( int x, int y)  /* Here number is a formal parameter  */
{
        return (x * y);
}
```

The above program passes actual arguments, **num1** and **num2** to the called function. The called function received the actual arguments, **num1** and **num2** in formal parameters, **x** and **y** respectively. The **multiply( )** function returns the multiplication of formal parameters **x** and **y**. This return value is collected by **result** variable in **main** (or **calling** function). At Thus, The value of **result** variable is printed.

2.5 RETURNING NON-INTEGER VALUES

In earlier section, we have discussed that functions return integer values. But there may be many situations where we have to return float, double, char type values. We can do this **explicitly** by changing the return type of the function. Let us consider, program 2.7 that multiples any kind of numeric value.

/* **Program 2.7** */

```
#include<stdio.h>
long double multiply (long double a, long double b );
int main()
{
        long double num1, num2, result;
        printf("Enter two numbers: ");
        scanf("%Lf %Lf", &num1, &num2);
        result = multiply(num1, num2);
        printf("Multiplication of %Lf and %Lf is %Lf\n", num1, num2,
        result);
        return 0;
}
/*  function definition  */
long double multiply(long double x, long double y)
{
        return (x * y);
}
```

In the above program, multiply( ) function has return type long double. It also received formal parameters as long double. We have also removed the restriction of positive integers in this program. The program correctly runs till the range of long double is not exceeded. The specifier %Lf is used for long double. Rest of the program is self explanatory.

2.6 FUNCTION ARGUMENTS

A function is requiredto use arguments so, it must declare variables that accept the values of the arguments. These variables are called the **formal parameters** of the function.The formal parameters are similar to other local variables inside the function and are created upon entry into the function and destroyed upon exit. During a call to a function, there are **two ways** that arguments can be passed to a function:

- Call by value
- Call by reference

2.6.1 Call by Value

The **call by value** method of passing arguments to a function **copies** the actual value of an argument into the formal parameter of the function. In this case, changes made to the formal parameter inside the function have **no effect** on the arguments of calling function.

By default, 'C' programming language uses **call by value** method to pass arguments. Thus, the code within a function **cannot** alter the arguments used to call the function, i.e. there is **no relation** between **actual** and **formal** parameter. Change informal parameter doesn't affect actual parameter. Consider the function swap() in Program 2.8. This function exchange the values of two numbers with each other.

```
/*  Program 2.8   */
#include<stdio.h>
void swap(int a, int b);        /*   function declaration */
int main ()
{
/* local variable definition: Note: variables a and b are local to this
block only */
int a = 100;
int b = 200;
printf("– – – – – – – – – – – – – – – – – – – – – – – – ");
        printf("Inside main: Before swap, value of a : %d\n", a );
        printf("Inside main: Before swap, value of b : %d\n", b );
printf("– – – – – – – – – – – – – – – – – – – – – – – – – – – ");
/* calling a function to swap the values */
swap(a, b);              /* function calling   */
printf("– – – – – – – – – – – – – – – – – – – – – – – – – – – ");
        printf("Inside main: After swap, value of a : %d\n", a );
        printf("Inside main: After swap, value of b : %d\n", b );
printf("– – – – – – – – – – – – – – – – – – – – – – – – – – – ");
return 0;
}
/* function definition to swap the values */
void swap(int a, int b)
{ /*   Note:  The formal parameters int a and int b are local to this
block only   */
int temp;
temp = a;                /* save the value of a */
a = b;                   /* put b into a */
b = temp;              /* put a into b */
printf("Inside swap: value of a : %d\n", a );
printf("Inside swap: value of b : %d\n", b );
return;
}
```

The output of the above program is:

```
-----------------------
Inside main: Before swap, value of a :100
Inside main: Before swap, value of b :200
-----------------------
Inside swap: value of a :100
Inside swap: value of b :200
-----------------------
Inside main: After swap, value of a :100
Inside main: After swap, value of b :200
-----------------------
```

In the above program, variables a and b defined in main( ) have only local scope within the main. The **actual arguments, a** and **b**, are passed to function swap(int a, int b). These actual arguments are received by **formal parameters a** and **b**. Do not confuse with the name of variables. They have same name, but their memory locations are different. The changes made to local variables (in this case, **formal** parameters **a** and **b**)**DO NOT** change other variables with the same name (i.e. variables **a** and **b** in main function).

Thus, the program swaps the numbers with in the swap function. Actually no swapping is done for actual arguments. The following observations are made from this example:

- However the actual and formal parameter names are **same**, still they have different memory locations.
- Any changes made to local variables do **not** change other variables with the same name.

2.6.2 Call by Reference

In the previous section, we have seen that if a variable in the main program is used as an actualargument in a function call, its value won't be changed no matter what is doneto the corresponding formal argument in the function.

The **call by reference** method of passing arguments to a function **copies** the **address** of an actual argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the function call. This means that **changes** made to the **formal parameter affect** the **actual parameter (argument)**.

In **call by reference**, the address of actual arguments is passed to the functions just like any other value. Hence, accordingly you need to declare the formal parameters of function as pointer types. Since the **actual argument** variable and the corresponding **formal parameter** pointer **refer** to the **same memory location.** So, changing the contents of the formal pointer change the contents of the actualargument variable, as in the following function swap(), which exchanges the values of the two integer variables pointed to by its arguments.

Let us call the function swap() with call by reference in the following program 2.9;

```
/* Program 2.9 */
/* function definition to swap the values */
void swap(int *x, int *y)
{
        int temp;
        temp = *x;          /* save the value at address x */
        *x = *y;            /* put y into x */
        *y = temp;          /* put x into y */
        return;
}
#include <stdio.h>
/* function declaration */
void swap(int *x, int *y);
int main ()
{
        /* local variable definition */
        int a = 100;
        int b = 200;
        printf("––––––––––––––––––––––––– ");
        printf("Before swap, value of a : %d\n", a );
        printf("Before swap, value of b : %d\n", b );
        /* calling a function to swap the values.
        * &a indicates pointer to a ie. address of variable a and
        * &b indicates pointer to b ie. address of variable b.
        */
        swap(&a, &b);
        printf("------------------------");
        printf("After swap, value of a : %d\n", a );
        printf("After swap, value of b : %d\n", b );
        printf("------------------------");
        return 0;
}
```

The output of the above program is:

```
--------------------
        Before swap, value of a :100
        Before swap, value of b :200
--------------------
        After swap, value of a :200
        After swap, value of b :100
--------------------
```

which shows that values are swapped inside the function. This is happened because addresses of actual arguments are passed to the function. These addresses of actual arguments are received by corresponding formal parameter pointers. Any operations through these formal parameter pointers will change the content of actual arguments.

2.7 RECURSION

Recursion is the process in which a function repeatedly **calls to itself** in order to perform calculations. The 'C' programming language supports recursion i.e. a function to call itself. But before using recursion, programmers must be careful to define an exit condition from recursion process,otherwise it will go in an infinite loop.

Recursive function is very useful to solve many mathematical problems like to calculate factorial of a number, generating Fibonacci series etc. Instead of using recursive function, programmers prefer iterative version of the function.

Let us evaluate a factorial of a given number. The following function calculates factorials recursively:

```
int factorial(int n)
{
        int fact;
        if (n<=1)
        fact=1;
else
        fact = n * factorial(n-1);
        return fact;
}
```

Let us check the program execution flow for n = 4. The function is first called with an argument 4. Since 4 is not less than 1, control transferred to else part. Here, the statement

fact = n * factorial (n -1);

is executed with n = 4. Therefore,

fact = 4 * factorial (3);

will be evaluated. The **rvalue** is a function call expression with argument 3. This call is than return back to the function with value 3. The next statement to be evaluated is:

fact = 3 * factorial (2);

This again calls the same function with argument value 2. So, again the next statement:

fact = 2 * factorial (1);

is evaluated. This time factorial(1) = 1 returns. Thus, complete sequence of calling factorial(4) is:

```
fact  = 4 * factorial (3)
= 4 * 3 * factorial ( 2 )
= 4 * 3 * 2 * factorial (1 )
= 4* 3 * 2 * 1 = 24.
```

Let us consider another program to compute Fibonacci series for a given number using recursion:

```
/*  Program 2.10  */
#include <stdio.h>
int fibonaci(int i)
```

```
{
        if(i == 0)
{
        return 0;
}
        if(i == 1)
{
        return 1;
}
        return fibonaci(i-1) + fibonaci(i-2);
}
        int main()
{
        int i;
        for (i = 0; i < 10; i++)
{
        printf("%d \t%n", fibonaci(i));
}
        return 0;
}
```

Execute the program and verify the results.

The output of the program is:

```
0    1    1    2    3    5    8    13    21    34
```

2.8 ARRAYS AS FUNCTION ARGUMENTS

Like any other 'C' variable, you can also pass the values of a one dimensional array to a function. To pass a single-dimension array as an argument in a function, you have to declare function formal parameter in one of following **three ways**. All these three declaration methods generate similar results because each tells the compiler that an integer pointer is going to receive. Similarly, you can pass multi-dimensional array as formal parameters.

Mehotd 1:

The formal parameters are pointers to receive address of an array.

It is convenient to **use pointers as formal parameters in the function**. Once the function has the base address of the array, it can use pointer arithmetic to work with all the array elements. Following skeleton shows the method 1 procedure.

```
void myFunction(int *param)
{
.
.
.
}
```

Method 2:

Here, the formal parameter is a sized array. The size of formal parameter array is same as an array used in calling function. This way, whole array is passed to the formal parameter. The skeleton of Method 2 is shown as:

```
void myFunction(int param[10])
{
.
.
.
}
```

Method 3:

Finally, the formal parameter is an unsized arrayas follows:

```
void myFunction(int param[])
{
.
.
.
}
```

Now let us consider the following function which takes an array as an argument along with another argument representing the size of an array. The function returns an average of the numbers passed through the array. The Program 2.11 is showing the working for the above said task.

```
/* Program 2.11 */
double getAverage(int arr[], int size)        /* function definition */
```

```
{       int i;
        double sum=0;
        for (i = 0; i < size; ++i)
{
        sum += arr[i];
}
return (sum/size);
}
#include <stdio.h>
double getAverage(int arr[], int size);        /* function declaration */
int main ()
{
/* an int array with 5 elements */
int num[5] = {10, 12, 13, 1, 5};
double avg;
/* pass pointer to the array as an argument */
avg = getAverage( num, 5 ) ;        /* function calling */
/* output the returned value */
printf( "Average value is: %f ", avg );
return 0;
}
```

The output of the above program is:

```
Average value is: 8.2
```

2.8.1 Pointers as Function arguments

Let us consider the previous example. Now, in it we change the formal parameter list. Instead of declaring array as formal parameter, we use **pointer** as a **formal parameter** in the fucntion. Thus, our function code looks like:

```
double getAverage(int *ptr, int size)        /* function definition */
{
int i;
double sum=0;
for (i = 0; i < size; ++i)
{
```

```
sum += *(ptr + i ); /*accessing array contents using indirection operator */
}
return (sum/size);
}
```

There is no need to change the main program. The main program passes **num** as an argument, which refers to the **base address** of an array. This address is received by **ptr** pointer. Now, after receiving the base address of an array, rest of the program works in the same way as program 2.11 does.

2.8.2 Return array from function

The 'C' language does not allow to return an entire array as an argument to a function.Still, you can return a pointer to an array by specifying the array's name without an index. If you want to return a one-dimension array from a function, you have to **declare** a function **returning a pointer** as in the following example:

```
int * myFunction()
{
. .....
. .....
}
```

Read the above function header as:

"Function **myFunction** takes **no formal parameters** and **return an integer pointer**"

Another important point is that you cannot return the address of local variable through pointers. 'C' language does not permit this operation. However, if you want to return the address of local variable, then you should declare a **local** scope of the variable extends up to the end of the function in which they are defined. Got it!

Now consider the following function which compute the square of 10 integer numbers (1 to 10) and stored them in an array. Finally the array is returned back to the calling function through the pointer. Note that, we have declared the array as static in the function, as shown in program 2.12:

```
/* Program 2.12 */
#include <stdio.h>
```

```
        int * getInput( )        /* getInput( ) returns pointer to int   */
{
        static int square[10];
        int i;
        for ( i = 0; i < 10; ++i)
{
        square[i] = ( i + 1) * (i + 1 );
        printf( "square[%d] = %d \n", i, square[i]);
}
        return square; /* returning array square.   Name of array
        represent address */
}
        /* main function to call above defined function */
int main ()
{
        /* a pointer to an int */
        int *p; /* pointer p is used to receive return address from
        get Input( ) */
        int i;
        p = getInput( );        /* function getInput( ) returns pointer
to int  */
        for ( i = 0; i < 10; i++ )
{
        printf( "*(p + %d) : %d \n", i, *(p + i));
}
        return 0;
}
```

The output of the above code is:

```
        square[0] = 1
        square[1] = 4
        square[2] = 9
        square[3] = 16
        square[4] = 25
        square[5] = 36
```

```
square[6] = 49
square[7] = 64
square[8] = 81
square[9] = 100
*(p + 0) : 1
*(p + 1) : 4
*(p + 2) : 9
*(p + 3) : 16
*(p + 4) : 25
*(p + 5) : 36
*(p + 6) : 49
*(p + 7) : 64
*(p + 8) : 81
*(p + 9) : 100
```

Check your progress 1

1. Consider the following program:

```
void main( )
{
char *k="xyz;
f(k);
printf("%s\n",k);
}f(char *k)
{
k = malloc(4); strcpy(k, "pq");
}
```

What will be the output?

2. Write a 'C' function to split the list in several sub-lists depending on the number of digitsrepresenting the integers i.e. single digit integers will form a list, double digit numbers will form another list and so on.

3. If a function is declared as **void fn(int *p)**, then which of the following statements is valid to call function **fn**?

(A) fn(x) where x is defined as **int x;**

(B) fn(x) where x is defined as **int *x;**

(C) fn(&x) where x is defined as **int *x;**

(D) fn(*x) where x is defined as **int *x;**

4. What is the following function computing? Assume a and b are positive integers.int fn(int a, int b)

```
{if (b == 0)
   return b;
else
   return (a * fn(a, b - 1));}
```

5. Write a function that satisfies the following definition:

```
int isSorted(int *array, int nElements)
{
   /*
      array is a pointer to an array of ints.
      nElements is the number of ints in the array.
      This function returns true if the array elements are sortedin ascending numerical order,
        and false otherwise.
      Example: suppose array contains {5, 9, 13}, and nElementsis 3.
         Then this function returns true since 5, 9, and 13 are
         inascending numerical order.
      */
  }
```

6. Write a function that satisfies the following definition:

```
void findPositives(double *array, int *nElements) {
/*
array is a pointer to an array of doubles.
*nElements is the number of doubles in the array.
This function removes all the non-positive elements fromarray, and returns, in *nElements, the number of elements remaining.
*nElements must be greater than or equal to zero.
Example: suppose array contains {1.0, 2.0, -1.0, 3.0}, and *nElementsis 4. Then, after calling findPositives, the first three elements ofarray will be {1.0, 2.0, 3.0} and*nElements will be 3.
*/
 }
```

7. What is wrong with the following function that is supposed to return the average of two integers? [Note: there may be more than one thing wrong].

```
int average(int i, int j)
```

```
{
    return (i + j)/2;
}
```

How would you fix the function?

2.10 SUMMARY

A program should be divided into a main module and its sub modules. Each module is then further divided into sub-modules (or pieces) until the resulting modules are basic. These smaller pieces sometimes called 'modules' or 'subroutines' or 'procedures' or functions. A 'C' program is made of one or more functions, one and only one of which must be named **main**. The execution of the program always starts with main, but it can call other functions, including library function such as **printf**() and **scanf**(), to do some part of the task. Generally, the purpose of a function is to receive zero or more pieces of data, operate on them, and return at most one piece of data. Every function has three components: definition, declaration, and function calling. A function can be categorized according to number of parameters present in it and return type. Functions communicate through each other either in downward, upward or in bi-directional mode. Call by reference is most efficient method as compared to call by value. Use of pointers makes the function programming most efficient.

UNIT 3: STRUCTURES, UNION, ENUM AND TYPEDEF

Structure

3.0 INTRODUCTION

While describing arrays, we have seen that arrays allow you to define type of variables that can hold several data items of the same type but **structure** and **unions** are another **user defined data types** available in 'C', which allows us to store mix data items of different types. The structures are used to represent a record, Let us assume that we have to keep track of books in a library. The several attributes which we have to track for each book are; Title, Author, Subject and Book ID. One possible way is to keep different arrays for each attribute and apply search operations on different attributes. But this is not the fruitful way because it complicates the whole tracking procedure. Instead, we can use structures in place of arrays. Therefore using the structure we have the ability to define a new data type considerably more complex than the types we have seen. A structure is a combination of several different previously defined data types, including basic data types, arrays, pointers and structure itself.

3.1 OBJECTIVES

After going through this unit, you should be able to:

- Declare, create and operate on instances of structures
- Declare, create and operate on unions

3.2 STRUCTURE DEFINITION

A**structure** is a data structure whose individual elements can **differ** in **type**. Thus, a single structure might contain integer elements, floating-point elements and character elements. Pointers, arrays and other structure can also be included as elements within a structure. The individual structure elements are referred to as **members** of the structure.

The structure declaration is somewhat more complicated than array declaration, since a structure must be defined in terms of its individual members.

The syntax for declaration is:

```
struct struct_name
{
        data_type  member_1;
        data_type  member_2;
              :
              :
        data_type  member_N;
};
```

In the above declaration, **struct** is a required keyword, **struct_name** is name that identifies structure of this typeand member_1, member_2, ...,member_N; are *individual member* declaration.The *individual members* can ordinary variables, pointers, arrays, or other structure.Each member definition is a normal variable definition, such as **int i;** or **float f;** or any other valid variable definition.The member names within a particular structure must be distinct from one another, though a member name can be the same as the name of a variable defined outside of the structure.

For example:

```
struct  student
{
        int roll_no;
        char name [25];
        float marks;
};
```

The above is a **declaration of structure data type** called **student**. It is **not** a variable declaration, but a **typedeclaration**.This is called user defined data type. It consists of three members, **roll_no** of

int type, **name** of **char** array, and **marks** of **float** type. No memory allocation takes place after declaring the structure type.At the end of the structure declaration, before the final semicolon, you can specify one or more structure variables but it is optional.

3.2.1 Structure Variable Declaration

The **members** (i.e, roll_no, name and marks) are **not** accessed directly. In order to access the members of structure, the structure **variable** has to be **declared first**. As soon as the structure variables are declared (or created), the memoryallocation takes place.

The syntax for variable creation is:

stroge_class strut struct_name , variable1,variable2.....variableN;

where **storage_class** is an optional storage class specifier.

For the above example of **student** structure, following are student structure variable declaration:

struct student Sameer, Swati, Ankur;

The above declaration creates three structure type variables, namely; ***Sameer, Swati*** and ***Ankur***.The memory allocation takes place as soon as the variable of structure is created. The memory allocation will contiguous.Thus, to know how much memory is allocated to the structure variable, you can use **sizeof()** operator. The program 3.1 shows how much memory is allocated to structure variables,Sameer, Swati, Ankur.

```
/* Program 3.1 */
#include<stdio.h>
  struct   student
    {
       int  roll_no;         /* int takes 2 byte  */
       char name [25];    /* char array of 20 elements takes 25 bytes  */
       float marks;          /*  float takes 4 byte   */
    };
main()
{
       struct student Sameer, Swati, Ankur;
       printf("The sizeof of Sameer is : %d\n", sizeof(Sameer));
       printf("The sizeof of Swati is : %d\n", sizeof(Swati));
       printf("The sizeof of Ankur is : %d\n", sizeof(Ankur));
}
```

The output of the above program is:

```
The size of of Sameer is : 31
The size of of Swati is : 31
The size of of Ankur is : 31
```

In the above program, we have define structure student. The structure consists of three members. The size of a structure depends upon type of each member. In this case, **roll_no** is **int**, which takes **2** bytes. Member **name** is an **array** of **25** elements of **char** type, and each char takes **1** bytes in memory, thus a total of **25** bytes are taken by name member, and finally **float** takes 4 bytes in memory. The total bytes allocated to**structure** variables are **31** bytes.

You can also **declare** a structure **type** and **variables simultaneously**. Consider the following structure representing playing cards.

```
structplaying_card {
        int p;
        char*suit;
} card1,card2,card3;
```

The above statement defined a **user defined data type**: **playing_cards** and also declare structure variables **card1**, **card2**, and **card3**. The memory allocated for structure playing_cards variables are 4 bytes. A variable p takes 2 bytes and since suit is a pointer, which takes 2 bytes of memory in a 16 bit machine. Thus total of 4 bytes are allocated for variables card1, card2 and card3. The number for bytes taken by pointer is machine dependent.

3.2.2 Initializing Structure Members

Structure members can be **initialized at the time of declaration**. This is analogous to the initialization of arrays i.e. the initial values are simply listed inside the pair of braces, with each value separated by a comma. But make sure that values are listed in the same order as they appear in the structure definition. The syntax for initializing the value to the members of structure is:

```
structure_variable_name.member_name1 = value;
structure_varibale_name → member_name2 = value;
```

Here, the dot operator (.) is used for all types of structure variable except pointer variable. The "dot" operator is called the member access operator.The arrow operator (   ) is used for only pointer variable of structure (discussed in next subsection).

The values to **member_name** can be supplied through number of ways:

- You can use normal scanf() to enter the values through keyboard.
- You can use strcpy()function to initialize an array values.

Following is an example for initialization in different ways:

```
struct student Sameer = {100,Sammer, 87}; /* initialization during
                                              declaration */
Swati.roll_no = 101;          /* initialize roll_no of variable Swati
                                 with 101 */
strcpy(Swati.name, "Swati");   /* initialization using strcpy( )
                                 string function */
Swati.marks = 97;
scanf("%d", &Ankur.roll_no); /* initialize using scanf. Input is taken
                                 from keyboard */
scanf("%s", Ankur.name);
scanf("%d", &Ankur.marks);
```

The structure initialization and declaration can be done in a single step as follows:

```
struct  structure_name
{
      data_type member1;
      data_type member2;
          :
          :
      data_type memberN;
}variabel1 = {list of value},variable2 = {list of
value},.........variableN = {list of value}
;
```

For example:

```
1.    struct student
  {
      int roll_no;
      char name[25];
      float marks;
  }   Sameer ={100, "Sammer",87}, Swati ={101, "Swati", 97}, student3,
      student4;
2.    struct student
  {
        int roll_no;
        char name[25];
        float marks;
 }
        S1[3]={ {4, "Manoj",2.1},{5, "Manoj",2.9},{61, "Manoj",4.26}};
```

In the first example, we create and initialize two variables, while we just create the two variables student3 and student4 and they are uninitialized.

In the second example, we create an array of structure variable of size 3 and initialize the array with the values as follows:

```
S[0]  = {4,              /* 4 is assigned to roll_no */
"Manoj",                 /* Manoj is assign to name[25]  */
2.1                      /*  2.1 is assigned to marks    */
        }
S[1]  =  {5,  "Manoj",2.9}
S[2]  =  {61,  "Manoj",4.26}
```

The same member names can appear in different structures. There will be no confusion to the compiler because when the member name is used it is prefixed by the name of the structure variable. For example, the member **name** is common in two different structures, however we can access them unambiguously by using their respective structure names with "**dot**" operator:

```
struct fruit {
        char *name;
        int calories;
} snack;
struct vegetable {
        char *name;
        int calories;
} din_course;
snack.name="banana";       /Note: name is accessed through snack
                            variable */
din_course.name="cheese";/*Note: name is accessed through
                            din_course variable */
```

3.2.3 Accessing Structure Members

To access any member of a structure except the pointer type, we use the member access operator (.), i.e dot operator and arrow operator

(   ) i.e. pointer operator. The member access operator is coded as a period between the structure variable name and the structure member that we wish to access. The syntax of member accessing is:

```
structure_varibale_name. member_name;


structure_varibale_name     member_name
```

The dot operator (.) is used for all types of structure variable except pointer variable.

The arrow operator (   ) is used for only pointer variable of structure (discussed in next subsection).

Consider an example of **student** structure. We can **access** members of **structure variables** through operator. The program 3.2 shows declaration, initialization and accessing of structure variables.

```
/*  Program 3.2  */
#include <stdio.h>
#include <string.h>
struct student
{
        int roll_no;
        char name[25];
        float marks;
};
int main( )
{       struct student Sameer; /* Declare Sameer of type student */
        struct student Swati;  /* Declare Swati of type student */
        struct student Ankur;  /* Declare Ankur of type student */

/* Structure variable initialization using assignment  */
        /* Sameer specification */
        Sameer.roll_no = 100;
        strcpy(Sameer.name, "Sameer");
        Sameer.marks = 87;

        /* Swati specification */
        Swati.roll_no = 101;
        strcpy(Swati.name, "Swati");
        Swati.marks=97;

/* Structure variable initialization using scanf function  */
        scanf("%d", &Ankur.roll_no); /* suppose input to roll_no is :
                                                102
        scanf("%s", Ankur.name);    /* suppose input to name[25] is:
                                                Ankur
        scanf("%d", &Ankur.marks);  /* suppose input to
                                                marks is :  67
/* Accessing or printing Sameer info */
printf( "Student Roll No:  %d\n", Sameer.roll_no);
printf( "Student Name : %s\n", Sammer.name);
printf( "Student Marks : %d\n", Sameer.marks);

/* Accessing or printing Swati info */
printf( "Student Roll No:  %d\n", Swati.roll_no);
printf( "Student Name : %s\n", Swati.name);
printf( "Student Marks : %d\n", Swati.marks);

/* Accessing or printing Ankur info */
printf( "Student Roll No:  %d\n", Ankur.roll_no);
printf( "Student Name : %s\n", Ankur.name);
printf( "Student Marks : %d\n", Ankur.marks);

return 0;
}
```

The output fo the following program is:

```
Student Roll No : 100
Student Name    : Sameer
Student marks   : 87
Student Roll No : 101
Student Name    : Swati
Student marks   : 97
Student Roll No : 102
Student Name    : Ankur
Student marks   : 67
```

3.3 STRUCTURES WITHIN STRCUTURES

We have already seen that structure members may be the variable of any data type. Thus, a structure can also be the member of a structure. By using this facility complex data types can be created. Suppose we need to make a structure that contained both date and time information. One way to achieve this would be to combine two separate structures i.e. one for the date and one for the time. For example,

```
/* defining struct date: Now date is a user defined data type */
        struct date {
        int month;
        int day;
        int year;
};
/* defining struct time: Now time is a user defined data type */
        struct time {
        int hour;
        int min;
        int sec;
};
/* defining struct date_time: date_time is a user defined data type.
The strucutere date_time consist of two variables, today of type
date and now of type time.*/
        struct date_time {
        struct date today;
        struct time now;
};
```

The above statement declares a structure whose elements consist of two other previouslydeclared structures.

3.3.1 Initializing Structures within Structures

The initialization of structure member can be done in the same way as we did previously. Initialization of date_time can be done as follows:

```
        struct date_time past = {{10,11,1978},{10,11,13}};
```

which sets the **today** element of the structure **past**to the eleventh ofOctober, 1978. The **now** element of the structure is initialized to tenhours, eleven minutes, thirteen seconds.

Each item within the structure can bereferenced using **dot** operator (or **arrow** operator), if required. For example,

```
++past.now.sec;
if (past.today.month == 11)
printf("Correct month! \n");
```

Consider another example which shows nested structure at work:

```
1.      struct record
 {
        int a,b;
        float c;
        char d;
 };


struct class
{
        int pointer;
        float q;
};
struct room
{
        struct  record S1,S2;
        struct  class c1 [2];
}
```

here the member of structure "**room**" is the variables of structure "**record**" and "**class**".

All the above three structure can also be written as:

```
2.      struct room {
                        struct  record
                        {
                                int a,b;
                                float c;
                               char d;
                        }S1,S2;
```

```
                struct class
                {
                        int pointer;
                        float q;
                }c1[2];
}R1;
```

Both the above definition is for **same** structure and only difference is in the way of writing. In order to access the member of the "**record**" or "**class**", which is the sub-member of"**room**"structure, you can use following syntax for accessing the sub member:

```
Structure_variable_name.member_name.sub_member


Structure_variable_name    member_nameàsub_member
```

where the **dot** and **arrow** operator are used on the basis of types of variable.

For example:

The **R1** variable is of **room** type. To access the **sub-member** of **R1**, we use following statements:

R1.S1.a	R1.S2.b	R1.C1[0].p
R1.S1.b	R1.S2.c	R1.C1[0].q
R1.S1.c	R1.S2.d	R1.C1[1].p
R1.S1.d	R1.S2.a	R1.C1[1].q

3.3.2 Some important properties of Structure

Let us now explore the properties of structure with a view of programming convenience. We would highlight these properties with suitable example:

Property 1:

"The value of a structure variable can be assigned to another structure variable of the same type using the assignment operator"

For example:

```
struct student
{
        Int roll_no;
        char name[25];
        float marks;
};
struct student S1={200, "Ravi", 90.2};     /* initialization of S1
                                              variable */
struct student S2;                     /* declaration of S2
                                          variable of student type   */

S2.roll_no=S1.roll_no;    /* by using these three statements, we are
                             copying the */
S2.name =S1.name;         /*  value of member of S1 variable to S2
                             variable of    */
S2.marks =S1.marks;       /*student structure.        */
```

The above assignment can also be done in a **single** statement as follows:

```
            S1 = S2;
```

Property 2:

"The values of a structure variable can be checked for equality or not equality to another structure variable of the same type using equality and not equality operator"

For example:

```
strut record
{
int a;
float b;
}S1={1,2.1},S2={3,4.2};
```

We have define record as a user defined data type and create two variables, S1 and S2 initialized with some default values. Now check whether S1 is equal to S2 or not.

```
 if(S1==S2)      /*  this will return zero i.e.;Condition is false. */
  {
        printf("Both structure variables are same\n");
  }
 S2=S1;      /* make them equal  */
if(S1 !=S2)    /* This will return zero because We are checking for
inequality */
   {
        printf("Both structure variables are not same\n");
}
```

3.4 STRUCTURES AS FUNCTION ARGUMENTS

You can pass a structure as a function argument in very similar way as you pass any other variable or pointer in the function. You can access structure variables in the similar way as you have accessed in the previous example. Like arrays, you can pass a structure variable to the function as an argument. As we have already seen that when we use call by value method, change made in structure variable in function definition does not reflect in original structure variable in calling function.Consider thefollowing Program 3.3 that passes structure student to the function **display( )**.

```
/*  Program 3.3  */
#include <stdio.h>
#include <string.h>
struct student
{
        int roll_no;
        char name[25];
        float marks;
};
/* function declaration */
void display(struct student stu);
int main()
{
  struct student s1;
  printf("Enter student's name: ");
  scanf("%s",&s1.name);
  printf("Enter roll number:");
  scanf("%d",&s1.roll_no);
  printf("Enter Marks:");
```

```
 scanf("%d",&s1.marks);
display(s1);            // passing structure variable s1 as argument
 return 0;
}
/* function definition */
void Display(struct student stu)
{
        printf("Output\nName: %s",stu.name);
        printf("\nRoll: %d",stu.roll);
       printf("\nMarks: %d",stu.marks);
}
```

The output for the following program is:

```
Name: Ashish
Roll : 202
Marks: 67
```

In the above program, function display( ) takes student type structure. In the main, we have taken following inputs from the user; roll_no, name and marks member of s1, which is a student type. The structure s1 is passed to display() function, where these values are printed.

3.5 POINTERS TO STRUCTURES

You can define pointers to structures in very similar way as you have defined pointer for any other variable as:

struct student *struct_pointer;

The above statement declares **struct_pointer** is a pointer to **student** data type. This means struct_pointer can only point to student data type. After this, you can store the address of a structure variable in the above defined pointer variable.

Let us consider again student structure:

```
struct student
{
        int roll_no;
        char name[25];
        float marks;
}student1, student2, *struct_pointer;
```

The above structure creates two variables student1 and student2. We also create a pointer ***struct_pointer** of **student** type.

To find the address of a structure variable, place the **address (&)** operator before the structure's name as follows:

struct_pointer = &student1;

To access the members of a structure using a **pointer** to that **structure**, you must use theàoperator as follows:

struct_pointer roll_no;

struct_pointer name;

struct_pointer marks;

The operator is known as **arrow** operator. This operator **canonly** be used with structure pointer variables.

You can also access structure variables using following statements:

(*struct_pointer).roll_no;

(*struct_pointer).name;

(*struct_pointer).marks;

The Parenthesesare required because preference of structure "**dot**" operator "." is higher than indirection operator "*".

Let us re-write the program 3.2. The revised program 3.4 using structure pointer is shown below:

```
/*  Program 3.4   */
#include <stdio.h>
#include <string.h>
struct student
{
        int roll_no;
        char name[25];
        float marks;
};
/* function declaration */
void display(struct student *stu);   /* display has one parameter:
student type pointer  */
int main()
```

```
{
        struct student s1;      /* declare studetn1 of type student */
  struct student s2;   /* declare studetn2 of type student */
/* student1 specification */
student1.roll_no = 100;
strcpy(student1.name, "Sameer");
student1.marks = 87;

/* student2 specification */
Student2.roll_no = 101;
strcpy(student2.name, "Swati");
student2.marks = 97;

/* displaystudent1 info by passing address of student1 */
 display(&student1);                /*  function call */
/* displaystudent2 info by passing address of student2 */
  display(&student2);               /*  function call */
  return 0;
}
/*  function definition  */
void Display(struct student *stu)
{       printf("\nRoll: %d",sturoll);
        printf("Name: %s",stuàname);
        printf("\nMarks: %d",stumarks);
}
```

The output for the following program is:

```
Roll : 100
Name   : Sameer
Marks  : 87
Roll: 101
Name   : Swati
Marks  : 97
```

3.6 UNIONS

The union is a special data type available in 'C' whose syntax look similar to structures, but act in a completely different manner. It enables you to store different data types in the same memory location. The declaration, initialization and accessing mechanism is same as we have seen for structures. The only difference is in term "**union**". You have to use **union** keyword **instead** of **struct**. You can define a union with many members, but **onlyone** member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multi-purpose and it exhibits the **difference** between **union** and **structure**.

To define a **union**, you must use the union statement in very similar way as you did while defining structure. The **union** statement defines a new data type, with more than one member for your program.

The union syntax is:

union ***union_name*** {

type1;

type2;

...

};

where **union_name** is the name of union data type, **type1**, **type2**, ... are 'C' data types.

Note: Once you declare a union variable, the amount of memor reserved is just sufficient to be able to represent the **largest member**. (Unlike a structure where memory is reserved for **all** members).

Let us define a union type named **record** which has the three members **i**, **f**, and **str**:

```
union record
{
        int i;
        float f;
        char str[10];
} record1;
```

Now a variable of **record** union type can store an integer, a floating-point number, or a string of characters. This means that a single variable i.e. same memory location can be used to store multiple types of data. You can use any built-in or user defined data types inside a union based on your requirement.

In the previous example, 10 bytes are set aside for the variable **record1** since a **char str[10]** will take up 10 bytes, an **int** take 2bytes and **float** takes 4 bytes.

Data actually stored in a union's memory can be the data associated with **any** of its members. But **only one** member of a union can contain valid data at a given time in the program.It is responsibility of programmer to keep track of which type of data has most recently been stored in the union variable.

Following program 3.5 display total memory size occupied by the above union:

```
/* Program 3.5 */
#include <stdio.h>
#include <string.h>
union record
{
        int i;
        float f;
        char str[10];
};
int main( )
{
        union record record1;
        printf( "Memory size occupied by record : %d\n", sizeof(record));
        return 0;
}
```

The output of the above program is:

```
Memory size occupied by record : 10
```

3.6.1 Accessing Union Members

Like structures, member of a union can be accessed with the help of member access operator **(.)**. You would use **union** keyword to define variables of union type. Following program 3.6 demonstrates the usage of member access operator for union:

```
/* Program 3.6 */
#include <stdio.h>
#include <string.h>
union record
```

```
{
int i;
float f;
char str[10];
};
int main( )
{
        union record record1;
        record1.i = 10;
        record1.f = 220.5;
        strcpy(record1.str, "Hello world");
        printf( "record1.i : %d\n", data.i);
        printf( "record1.f : %f\n", data.f);
        printf( "record1.str : %s\n", data.str);
        return 0;
}
```

The output of the above program is:

```
record1.i : 7524
record1.f : 65203075.000000
record1.str : Hello world
```

The above output indicates that values of **i** and **f** members of union got junk values because final value assigned to the variable **str** has occupied the memory location and this is the reason that the value of **str** member is getting printed very well.

Let us take another example as shown in program 3.7 where we will use one variable at a time, doing so gives you more accurate idea about union data type.

```
/*  Program 3.7   */
#include <stdio.h>
#include <string.h>
union record
{
        int i;
        float f;
        char str[10];
};
```

```
int main( )
{
        union record record1;
        record1.i = 10;
        printf( "record1.i : %d \n", data.i);
        record1.f = 220.5;
        printf( "record1.f : %f \n", data.f);
        strcpy(record1.str, "Hello world");
        printf( "record1.str : %s \n", data.str);
        return 0;
}
```

The output of the above program is:

```
record1.i : 10
record1.f : 220.500000
record1.str : Hello World
```

From the above output, we can see that all the members are getting printed very well because one member is being used at a time.

3.7 ENUMERATED DATA TYPE

The 'C' language provides another user defined data type known as "enumerated data type". The use of this data type is to make program more readable. You can create your own data type with predefined values, if we know in advance the finite set of values that a data type will have.

The syntax for creating enumerated data type is as follows:

enum *identifier* { val1, val2, ... valN };

where *identifier* represents the user defined enumerated data type and val1, val2, ... valN are called **members** or **enumerators**. The **values** mentioned within the braces are **not** variables, but in fact are **constant** values that the **enum** can take. The **enumerators** or **members** are automatically assigned values starting from 0 to n-1. The assignment, arithmetic and comparisons operations are allowed on enumerated type variables.

For example:

enum MONTHS {Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec};

In the above statement, **MONTHS** is a **user defined data type** which take values from the above **set** consisting of {Jan, Feb, mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec}.

We can declare variables using the following syntax:

enum identifier var1, var2, ... varN;

the variables var1, var2, ...varN can take values only from the set { val1, val2, ... valN }.

For example:

enum MONTHS month1, month2;/* variables declared of type months */

In above statement, **month1** and **month2** are variables of type **MONTHS**.

We can assign values to the variables as:

```
month1=Jan;
month2=Aug;
```

The statement:

```
printf("%d\n", month2 – month1 );
```

will print 7, because integer value for Aug is 7 and that of Jan is 0.

The following statement will also work:

```
        if(month1 < month2)
        {
-----
-----
        }
        else
{
-----
-----
}
```

You can also change the default value by assigning the interger values to the enumerators when you create the data type:

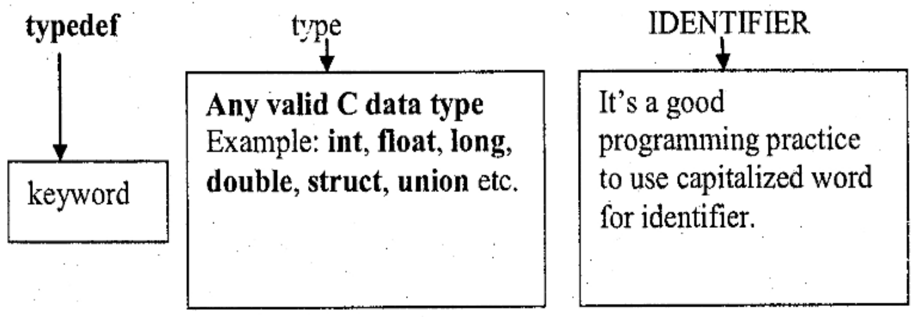
For example:

enum MONTHS {Jan =1 , Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec};

Now the enumerators will have values starting from 1 to 12

3.8 TYPE DEFINITION

The 'C' programming language provides, **typedefinition**, using a keyword called **typedef** which you can use to give a type a new name. The syntax for type definition is:


typedef
type
IDENTIFIER
keyword
Any valid C data type
Example: int, float, long, double, struct, union etc.
It's a good programming practice to use capitalized word for identifier.

Following is an example to define a term REALNUMBERS for all real numbers.

typedef long double REALNUMBERS;

After this type definitions, the identifier **REALNUMBERS** can be used as an abbreviation for the type **long double.**

for example; REALNUMBERS r1, r2;

It's a good programming practice to use uppercase letters for these definitions to recall the user that the type name is really a symbolic abbreviation, but you can use lowercase, as follows:

typedef long double realnumbers;

The **typedef** can be used to give a name to **user defined data type** as well. For example, we can use **typedef** with structure to define a new data type and then use that data type to define structure variables directly. The following program 3.8 shows how to use **typedef** in structures.

```
/*  Program 3.8  */
#include <stdio.h>
#include <string.h>
typedef struct student
{
int roll_no;
char name[25];
float marks;
}STUDENTS;
int main( )
{
    STUDENTS Student; /* Declare Sameer of type student */
    Student.roll_no = 100;
strcpy(Student.name, "Sameer");
Student.marks = 87;
printf( "Student Roll No: %d\n", Student.roll_no);
printf( "Student Name : %s\n", Student.name);
printf( "Student Marks : %d\n", Student.marks);
return 0;
}
```

The output of the above program is:

```
Student Roll No: 100
Student Name  : Sameer
Student Marks  : 87
```

Check your progress 1

1. What's the difference between these two declarations?

   ```
   struct x1 { ... };
   typedef struct { ... } x2;
   ```

2. Why doesn't

   ```
   struct x { ... };
   x thestruct;
   ```

   work?

3. Define a suitable data structure to store the information like student name, roll number,enrolment centre and marks of five different subjects. Write a 'C' function to insertsufficient data in your data structure and function to print the name of the student andthe total obtained marks who have secured highest total marks for each and every enrolment centre

4. What will be the output of the following code segment, if any?

   ```
   myfunc ( struct test t) {
   strcpy(t.s, "world");
   }       main( ) {
           struct test { char s[10]; } t;
           strcpy(t.s, "Hello");
           printf("%s", t.s);
           myfunc(t);
           printf("%s", t.s);
   }
   ```

5. What will be output of following 'C' code?

```
void main()
{
 struct field
{
int a;
char b;
}bit;
struct field bit1={5,'A'};
char *p=&bit1;
*p=45;
clrscr();
printf("\n%d",bit1.a);
getch();}
```

6. What will be output of following 'C'code?

```
void main()
{
struct india
{
char c;
float d;
};
struct world
{
int a[3];
char b;
struct indiaorissa;
};
struct world st ={{1,2,3},'P','q',1.4};
printf("%d\t%c\t%c\t%f",st.a[1],st.b,st.orissa.c,st.orissa.d);
getch();
}
```

3.9 SUMMARY

Structures are the special feature of 'C' language that permits mixing of different data types related in some logical sense with a single entity. A **struct** variable has members which can be accessed by the **dot** and **arrow** operators. A union is a type similar to the **struct** that can hold at any time just one of its members, which may be of various types. In this Unit, we also have seen the user defined data type, **enum** and **typedef**. The **typedef** makes an alias to the pre existing data type, while **enum** provide the flexibility to create your own data type.

UNIT 4: FILE AND MEMORY MANAGEMENT IN 'C'

Structure

4.0 INTRODUCTION

So far, all the output (formatted or not) in this course has been written out to with**standard output** (which is usually the monitor). Similarly all input has come from **standard input** (usually associated with the keyboard).The keyboard and terminal are usually console oriented I/O functions and poses following problems:

- If same data is to be processed later on, than it has to be re-enter again, which is a cumbersome process.
- Entering large volume of data from keyboard is very time consuming.

The 'C'language considers all the devices as files. Thus, devices such as the keyboard and screen display are treated in the same way as files. There are following three files **automatically** opened by your operating system when a program executes to provide access to the keyboard and screen.The **filepointers**are the way to access the file for reading and writing purpose.

Standard File	File Pointer	Device
Standard input	stdin	Keyboard
Standard output	stdout	screen
Standard error	stderr	Your screen

In this Unit, we are going to describe the storage and retrieval of data from the files which overcomes the handling of large volume of data from the keyboard at run time. The 'C' programmer can also read data directly from files and write directly to files.

4.1 OBJECTIVES

After going through this unit, you should be able to:

- Declare files and perform file I/O
- Know various file operation modes
- **fscanf( )** and **fscanf( )** formatted I/O operations
- Dynamic allocation of memory

4.2 FILES

This Unit describes that how 'C' programmers can create, open, and close text or binary files for their data storage. A **file** represents a sequence of bytes, or collection of related data treated as a unit, does not matter if it is a text file or binary file. File is also defined as a permanent storage of data referenced by a name. Usually, files are accessed sequentially, i.e., any byte can be changed. The prime purpose of a file is to keep the record of data. Since the contents of primary memory are lost when the computer is shut down, we need files to store our data in a more permanent form. 'C' programming language provides access of high level I/O functions as well as low level (OS level) calls to handle file on your storage devices. This Unit will take you some of the most important high level I/O functions for the file management, listed in Table 1.

The DOS I/O redirection enables you to use redirection operators for reading and writing of files. Suppose you have to read the data from the file **input.txt** and want to write the content after processing to **output.txt**. Thenfollowing command:

```
C:\ >check  <input.txt  >output.txt
```

will run the program **check**, and:

Here, input will be read from the file **input.txt** (or any other filename) instead of the keyboard, and output will be written to the file **output.txt** instead of the display. The operators **<** (**less than** symbol) and **>** (**greater than** symbol) are used as input and output redirection respectively.

There are so many circumstances when a program has to read/write directly to a file, and also perform some operations that can be done by standard library routines available in 'C' language.

To work with files, the following steps must be taken:

1) Declare **FILE type variables**.
2) Connect the **FILE** variable with the actual data file on your harddisk. Theconnection of a **FILE** variable with a file name is done with the**fopen()** function.
3) Perform I/O with the actual files using **fprint()** and **fscanf()**functions.
4) Disconnect the connection between the **FILE** variable and actual disk file. This disassociation is done with the **fclose()** function.

Table 1: High level I/O operations

Function name	operation
fopen()	· Creates a new file for use
	· Opens an existing file for use
fclose()	· Closes a file which has been opened for use
getc()	· Reads a character from a file
putc()	· Writes a character to a file
fprinf()	· Writes a set of data values to s file
fscanf()	· Reads a set of data values from a file
getw()	· Reads an integer from a file
putw()	· Writes an integer to a file
fseek()	· Sets the position to a desired point in the file
ftell()	· Gives the current position in the file (in terms of bytes from the beginning)
rewind()	· Sets the position to the start of the file

4.3 FILE Pointer Variable

Now, before using the high level functions in your program, you must include them into your program. This can be done by including the file:

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

as the first statement in your program.

The first step to use files in 'C' programs is to declare a **FILE** variable. **FILE** is a defined data type in **stdio.h** file. The variable of **FILE** type must be declared before you use files (which is a predefined type in 'C') and it is a pointer type variable. For example, the following statement

FILE *infile, *outfile;

declares the variable **infile** and **outfile** to be a *"pointer* to type **FILE"**. The * must be repeated for each variable.

4.4 OPENING A FILE

Hence, after declaring the **FILE** variable, and before using it, you must connect it with a specific file name. The **fopen()** function performs this connection. You can use the **fopen( )** function to **create** a new file or to **open** an existing file, this call will initialize an pointer variable of the type **FILE**, which contains all the information necessary to control the stream. Following is the prototype of this function call:

FILE *fopen(const char * filename, const char * mode);

Here **filename** is a string literal which you can use to name your file, or it may be the pathname of the disk file.

The access **mode** indicates how the file is to be used.Access mode could be a string variable and have one of the following values:

Mode	**Description**
File Category:	**Text files**
r	Opens an existing text file for reading purpose.
w	Opens a text file for writing, if it does not exist then a new file is created. Here your program will start writing content from the beginning of the file.
a	Opens a text file for writing in appending mode, if it does not exist then a new file is created. Here your program will start appending content in the existing file content.
r+	Opens a text file for reading and writing both.
w+	Opens a text file for reading and writing both. It first truncate the file to zero length if it exists otherwise create the file if it does not exist.
a+	Opens a text file for reading and writing both. It creates the file if it does not exist. The reading will start from the beginning but writing can only be appended.
File Category: Binary files To handle binary files you can use below mentioned access modes instead of the above mentioned accessed modes. Here, **b** stands for **binary** files; whereas **r, w, a** has usual meaning as described above.	
"rb", "wb", "ab", "ab+", "a+b", "wb+", "w+b", "ab+", "a+b"	

Text files store data as a sequence of characters,whereas binary files store data as they are stored in primary memory.

Now, let us consider a program segment that shows how to open a file in write mode:

```
char filename[80];
FILE *outfile;
printf("Enter the name for new file:");
gets(filename);
outfile = fopen(filename,"w");
```

In the above program segment, **outfile** is declared as FILE type pointer. The function gets( ) is used to take the name of file, since name of file is a string literal. Finally, **filename** and "**w**" mode is passed as an argument to **fopen**( ) function. The function fopen( ) returns a pointer to the outfile FILE pointer.

Next, consider the following statement

```
char filename[80];
FILE *infile;
infile = fopen("myfile.txt","r");
```

connects the variable **inffile** to the disk file **myfile.txt** for **read** access. Thus, **myfile.txt** will **only** read. In this example, we explicitly pass the name of file to be read.

4.5 READING AND WRITING TO FILES

Once a file has been created or opened, data can be read from it as well as you can write the data to the file. The reading and writing data can be done by using the functions listed in Table 1. In this section, we will discuss these functions one by one.

4.5.1 getc and putc function

The functions fgetc( ) and fputc( ) are simplest functions for I/O operations. The **fgetc()** function reads a character from the input file referenced by **FILE** pointer **fin**. The return value is the character read, or in case of any error it returns EOF. The **fgetc( )** is used when **fin** pointer opens a file in read "**r**" mode. The syntax for **fgetc( )** is as follows:

```
int fgetc( FILE * fin );
```

Conversely, the function **fputc( )** writes the character value of the argument **ch** to the output stream referenced by **fout**. It returns the written character on success otherwise EOF if there is an error. The **fputc( )** is used when **fout** pointer opens a file in read "**w**" mode. The syntax for **fputc( )** is given below:

```
int fputc( int ch, FILE *fout );
```

The program 4.1 demonstrate the use of **fputc( )** and**fgetc( )**.This program declares two pointers **fin** and **fout** for reading and writing to the files respectively. The getc( ) function reads a character from the input file test.txt and putc( ) function writes that character on to out.txt file.

```
/* Program 4.1 */
#include <stdio.h>
void main()
{
        FILE *fin, *fout;
        char ch;
fin = fopen("test.txt", "r");   /* opening file test.txt in read mode */
fout = fopen("out.txt", "w");   /* opening file out.txt in write mode
while((ch = getc(fin) != EOF)  /* read the characters till end of file is
                                        reached */
fputc(ch, fout);  /* write the character into output stream pointed by
                        fout */
fclose(fin);      / close the file pointers */
fclose(fout);
}
```

4.5.2 fgets( ) and fputs( ) function

Sometimes we need to read a string from a stream, this can be done with the help of file function fgets( ). The syntax of fgets is as follows:

char *fgets(char *buff, int n, FILE *fin);

The functions **fgets**() reads up to **n - 1** characters from the input stream referenced by **fin**. It copies the read string into the buffer **buff**, **appending** a **null character** to terminate the string. If **fgets**( ) encounters a newline character **'\n'** or the end of the file **EOF** before they have read the maximum number of characters, then it returns only the characters read up to that point including new line character.

Similarly, to write a null-terminated string to the output stream, you can use the fputs( ) function. The syntax of fputs( ) is as follows:

int fputs(const char *str, FILE *fout);

The function **fputs**() writes the string **str** to the output stream referenced by **fout**. It **returns** a non-negative value on success, otherwise EOF is returned in case of any error.

The program 4.2 demonstrate the use of **fgets( )** and**fputs( )**.In this program, we use standard I/O file pointers, stdin for keyboard and stdout for screen. These pointers receive the input from keyword and send the output to the screen respectively. On pressing the Enter key at the beginning of a line, fgets() reads the newline and places it into the first element of the array line. Thus, condition inside while loop becomes FALSE, and loop terminates. Encountering end-of-file also terminates it.

```
/* Program 4.2 */
/* parrot.c — using fgets() and fputs() */
#include <stdio.h>
#define MAXCHAR 20
int main()
{
  char line[MAXCHAR];
  while (fgets(line, MAXCHAR, stdin) != NULL &&line[0] != '\n')
  fputs(line, stdout);
  return 0;
}
```

The output of the following program is:

```
Hello how are you
Hello how are you
I am fine!. What about you?
I am fine!. What about you?
[enter]
```

The program gives correct ouput. This is because the second line contains 27 characters, and the line array holds only 20, including the newline character. When fgets() read the second line, it read just the first 19 characters, upto the b in the word *about*. They were copied into line, and printed on the screen by using the function fputs( ). Since fgets() couldn't reached to the end of a line, line did not contain a newline character, so fputs() didn't print a newline. The third call to fgets() res-started where the second call left off. Therefore, it read the next 19 characters into line, beginning with the *o* after the *b* in *about*. This next block replaced the previous contents of line and printed on the same line. Thus, fgets() read the second line in blocks of 19 characters, and fputs() printed it in the same-size blocks.

4.5.3 fprintf() and fscanf() Functions

The file I/O functions **fprintf**() and **fscanf**() work just like **printf**() and **scanf**(), except that they require an additional **first** argument to identify the proper file. You've already used fprintf(). The general format for fprintf( ) is:

fprintf(fp, "format string", list);

where fp is a file pointer associated with a file that is openend for writing. The format string contains output specification for the items in the ist. The list may include variables, constants and expressions.

For example:

fprintf(fout, "%s %d", str, t); /* write to file */

here, **str** is a variables of type **char** and **t** is a **int** variable.

The general format of **scanf** is:

fscanf(fp, "format string", list);

Here, reading of the items is dome from file specified by fp, as per specifications contained in the format string. Now let us consider the following example:

fscanf(fin, "%s%d", str, &t); /* read from file pointed by fin */

fscanf also returns number of items that are successfully read and when end of file is reached, it returns the value of EOF.

Another useful function for file I/O is **feof()** which tests for the end-of-file condition. **feof** takes one argument, the FILE pointer, and returns a nonzero integer value (TRUE) if an attempt has been made to read the end of a file. It returns zero (FALSE) otherwise.

Following program segment shows one of its use:

```
if (feof(infile))            /* infile is FILE pointeer  */
            printf ("No more data \n");
```

Program 4.3 illustrates both of these file I/O functions, along with the rewind() function.

```
/* Program 4.3 */
#include <stdio.h>
#include <stdlib.h>
#define MAX 40
int main(void)
{
  FILE *fout;
  char words[MAX];
  if ((fout = fopen("test.txt", "a+")) == NULL)
   {
    fprintf(stdout,"Can't open \"test\" file.\n");    exit(1);
   }
    puts("Enter words to add to the test file; press Enter key for
terminate:");
```

```
  while (gets(words) != NULL  && words[0] != '\0')
    fprintf(fout, "%s ", words);
  puts("File contents:");
  rewind(fout);                           /* go back to beginning of file */
  while (fscanf(fout,"%s",words) == 1)
    puts(words);
  if (fclose(fout) != 0)
    fprintf(stderr,"Error closing file \n");
  return 0;
}
```

The above program enables you to add words to a file. By using the "a+" mode, the program can both read and write in the file. The first time the program is used, it createsthe **test.txt** file and provide a way to place words in it. When you use the program subsequently, it enables you to add (append) words to the previous contents. The append mode addsthe content to the end of the file, but the "a+" mode also provide you a way to read the whole file. The rewind() fucntion takes the program to the file beginning so that the final while loop can print the file contents. On pressing the Enter key, gets() places a null character in the first element of the array, forcing the program to terminate the loop.

The **rewind( )** function takes a file pointer and resets the position to the start of the file.

4.6 FILE STATUS FUNCTIONS

Sometimes, we need to know whether file has reached to EOF marker or some error occurred. To know this, we use following three functions:

- **feof( )**
- **ferror( )**
- **clearerr( )**

The functions, **feof( )** and, **ferror( )** determine if a file has reached to end-of-file or if an error has occurred. The general syntax for **feof( )** and **ferro( )** is as follows:

int feof(FILE *fp);

int ferror(FILE *fp);

Each FILE pointer which you use to read and write data from and to a file contains flags that the system sets when certain events occur.

On getting some error, it sets the error flag; otherwise if you normally reach to the end of the file during a read, it sets the EOF flag. The functions **feof()** and **ferror()** give you a simple way to test these flags, i.e. they'll return non-zero (TRUE) if they're set.

On the other hand, once the flags are set for a particular stream, they stay in the same state until you call **clearerr()** to clear them. The general syntax for **clearerr( )** is as follows:

void clearerr(FILE *fp);

The following program shows the use of above functions:

```
/* Program 4.4 */
// read binary data, checking for eof or error
int main()
{
   int a;
   FILE *fp;
  fp = fopen("binaryints.dat", "rb");
  // read single ints at a time, stopping on EOF or error:
  while(fread(&a, sizeof(int), 1, fp), !feof(fp) && !ferror(fp))
{
     printf("We read %d\n", a);
   }
  if (feof(fp))
    printf("End of file was reached.\n");
  if (ferror(fp))
    printf("Some error occurred.\n");
  fclose(fp);
   return 0;
}
```

4.7 RANDOM ACCESS TO FILES

Once the files have been opened, we need some times to access the specific part of a file. This can be done by positioning the file pointer directly to the desired position in the file. This can be achieved with the help of library functions **fseek( )**. The **fseek**() function treats a file like a byte array and move directly to any particular byte in a opened file by the function **fopen( )**. The syntax of **fseek**( ) is as follows:

int fseek(fp, ± offset, mode);

The **fseek**() has three arguments and **returns** an **int** value.

The first argument to fseek() is a **FILE** pointer, **fp**, to the file being searched.

The second argument to fseek() is called the ***offset** (or **numberofbytes**)*. This argument tells how far to move from the starting point (as shown in Table 2). The argument must be a integer value. It can be positive (move forward), negative (movebackward), or zero (stay on same position).

The third argument is the **mode**, and it identifies the starting ~int.

Table 2: Mode for fseek

Mode	Measures Offset From
SEEK_SET	Beginning of file
SEEK_CUR	Current position
SEEK_END	End of file

Following are some examples of **fseek**( ) function calls, where **fp** is a file pointer:

```
fseek(fp, 0, SEEK_SET);  // go to the beginning of the file
fseek(fp, 10, SEEK_SET); // go 10 bytes into the file
fseek(fp, 2, SEEK_CUR);  // advance 2 bytes from the current position
fseek(fp, 0, SEEK_END);  // go to the end of the file
fseek(fp, -10, SEEK_END); // back up 10 bytes from the end of  the file
```

The value **returned** by **fseek**() is **0** if everything is okay, and **-1** if there is an **error**, such as attempting to move past the bounds of the file.

The **ftell**() function is type **long**, and it returns the current file location.**ftell**() specifies the file position by returning the number of bytes from the beginning, with the first byte being byte 0, and so on. The syntax of ftell( ) is as follows:

long ftell(fp);

Let us consider the statement

fseek(fp, 0L, SEEK_END);

It sets the position to an offset of 0 bytes from the file end. That is, it sets the position to the end of the file. Next, the statement

last = ftell(fp);

assigns to last the number of bytes from the beginning to the end of the file.

Next consider the following loop:

```
for (count = 1L; count <= last; count++)
{
```

```
            fseek(fp, -count, SEEK_END);        /* go backward */
            ch = getc(fp);
}
```

The first iteration positions the program at the first character before the end of the file (that is, at the file's final character). Then the program prints that character. The next loop positions the program at the preceding character and prints it. This process continues until the last character is reached and printed.

4.8 COMMAND LINE ARGUMENTS

'C' language provides the convenient way to pass some values from the command line (or terminal or console) to your 'C' programs when they are executing. These values are called **commandline arguments** and they are important in many times for your program especially when you want to run your program from outside instead using programming those values inside the code.

The command line arguments are handled using main() function arguments where **argc** refers to the number of arguments passed, and **argv[ ]** is a pointer array which points to each argument passed to the program. The following program 4.5 checks if there is any argument supplied from the command line and take action accordingly. Assume that filename is test.exe.

```
/* Program 4.5 */
#include <stdio.h>
int main( int argc, char *argv[] )
{
        if( argc == 2 )
        {
                printf("The argument supplied is %s\n", argv[1]);
        }
        else if( argc > 2 )
        {
                printf("Too many arguments supplied.\n");
        }
        else
        {
                printf("One argument expected.\n");
        }
        }
```

Run the program from command prompt like this:

```
C:\>test.exe testing
```

The output of the following program is:

```
The argument supplied is testing
```

Again, **run the program from command prompt like this:**

```
C:\>test.exe testing1 testing2
```

The output of the following program is:

```
Too many arguments supplied.
```

Finally, **run the program from command prompt like this:**

```
C:\>test.exe
```

The output of the following program is:

```
One argument expected
```

You must remember that that **argv[0]** holds the **name** of the program itself and **argv[1]** is a pointer to the **first command line argument** supplied, and ***argv[n]** is the **last** argument. If no arguments are supplied, **argc** will be one, otherwise if you pass one argument then **argc** is set at 2.

4.9 MEMORY MANAGEMENT

Dynamic memory allocation is required when we need to manage available memory. For example, it is a common problem that we do not know how large to make arrays when they are declared. This is because many times the memory allocation decisions are made during the runtime. Since 'C'language does not support automatic garbage collection, it is very necessary to manage alldynamic memory used during the program execution. Consider an example where you want to monitor the student information that is stored in a structure. We also require that program should be general purpose so that array is sufficient to hold the biggest possible class size of students:

```
struct student class[500];
```

But what happens when there are only ten students in a class. In that case, the above statement is infeasible and a huge amount of memory is wasted especially because the **student** structure is too large.

This fact forces to use and **create correct-sized array variables during runtime**. The 'C' language provides users a flexibility to dynamically allocate and deallocate memory when required. The **<stdlib.h>** provides four functions that can be used for dynamic memory allocation. The Table 3 provides the following four functions:

Table 3: Dynamic memory Allocation and Releasing Function

Function prototype	Description
void *calloc(int num, int size);	This function allocates an array of **num** elements each of whose size in bytes will be **size**. The array is initialized to zeros.
void free(void *address);	This function releases a block of memory block specified by **address**.
void *malloc(int num);	This function allocates an array of **num** bytes and leaves them **uninitialized**.
void *realloc(void *address, int new size);	This function re-allocatesmemory extending it upto **newsize**.

4.9.1 calloc( ) function

The **calloc** function is used to **allocate memory to a variable during the program execution**. The function takes two arguments that specify the number of elements to be reserved, **num**, and the **size** of each element in bytes (obtained from **sizeof**). The calloc function returns a pointer to the beginning of allocated memory block. The storage area is also initialized to zeros.

The program 4.6 allocates dynamic memory required for **10** integers as follows.

```
/* Program 4.6 */
#include<stdio.h>
#include <stdlib.h>
main( )
{
        int* A = NULL;          /* create int pointer and set to NULL. */
        if ((A = calloc(10, sizeof(int)*n)) != NULL)
        {
                for (i =0;i <n;i++)
                *(A+i) = i;     // you can replace *(A+i) by A[i] if you wish
        }
        else
        {       printf("calloc failed: Exiting Program! \n \n");
        exit( -1 );
        }
}
```

The above program creates a **int** type pointer variable A and set it to **NULL**. The **if** condition is checked against that whether memory block is allocated or not. If allocated properly, the condition becomes true and **for** loop allocates index value as content to memory block pointed by **int** pointer A. If condition of if statement becomes false, then program reports an error message by **exit(-1)**.

4.9.2 malloc( ) function

The **malloc** function allocates a memory block of size **n**bytes . The **malloc** function returns a pointer **(void*)** to the contiguous block ofmemory which is uninitialized. This **void*** pointer can be used for any pointertype. If enough contiguous memory is not available, then mallocreturns NULL, so be sure to check that memoryallocation was successful by using the following statement:

```
void* ptr;
if ((ptr=malloc(n)) == NULL)
exit(-1);
else
{
Printf("Memory allocation successful!");
}
```

Consider the program 4.7 where you do not know the length of the text you need to store, for example if you want to store a detailed description for a person, the youWe need to define a pointer to the character without defining how much memory is actually required and later at some satge we can allocate memory as shown below in the proram:

```
/* Program 4.7 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main()
{
char name[100];
char *detail;
strcpy(name, "Sameer Singh");
/* allocate memory dynamically */
detail = malloc( 100 * sizeof(char) );
if(detail== NULL )
{
        fprintf(stderr, "Error - unable to allocate required memory\n");
}
else
{
        strcpy(detail, "Sameer Singh is resident of M.P. ");
}
        printf("Name = %s\n", name );
        printf("detail: %s\n", detail);
}
```

The output of the following program is:

```
Name = Sameer Singh
Description: Sameer Singh is resident of M.P.
```

4.9.3 realloc() function

realloc( ) is another useful function for situations where we need toresize an existing memory block. To reallocate a newblock, we must copy the old values to the new block and then freethe old memory block. The syntax of **realloc( )** is:

void *realloc(void *ptr, size_t size);

The realloc() function changes the size of the memory block pointed toby ptr to **size** bytes. The contents of the old block are unchanged and newly allocated memorywill be alloacated and it is uninitialized. If ptr is NULL, resizing fails otherwise it is successful.

4.9.4 free( ) function

This function is used to free the occupied memory. When the **variables are not required**, the space allocated to them by malloc, **calloc and realloc** should be returned to the system. This is done by,

free(address);

that will cause the program to give back the block to the heap (or free memory). The argument to free is any address that was returned by a prior call to malloc, **calloc and realloc.**

Check your progress 1

1. Write a program to copy the contents of one file to another.
2. Two text files text1.txt and text2.txt contain sorted lists of integers. Write as program to create a thirds fie text3.txt which contains a single sorted, merged list of these two lists. Your program should use command line arguments.
3. Create a file to store and display the content.

4. Read a text file and create another file in which multiple blanksare replaced by single blank.
5. Create a file to store integers and display the contents.
6. Write a line of text in a file and display the contents.
7. Write a program to find the size of file in bytes (opened in text mode).
8. Write a program to find the size of file in bytes (opened in binary mode).
9. Write a program to find sum and average of n integer numbers using command line arguments.
10. Write a program to copy the contents of one file to another file using command line arguments
11. Use command line arguments to pass an integer and display it in reverse form using recursive function.
12. Use command line arguments to sort a list of given numbers. All numbers must be read as command line arguments.
13. What is the output of the following program segment.

```
while((ch=fgetc(fptr)) != EOF)
{
        If((ch >= 'A') && ( ch <= 'Z'))
        ch +=32;
        fputc(ch, stdout);
}
```

14. Write a program tocount number of words, characters, blanks, punctuation marks in a file.

4.9 SUMMARY

In this Unit, we have studied that how external files may read and write by 'C' programs. File I/O functions are very similar to the console I/O functions that we have studied in previous Units, but high level I/O functions are very useful for handling large volume of data. The dynamic memory allocation functions are also studied with examples. The usefulness of these functions are identified.

UNIT 5: PREPROCESSOR DIRECTIVES AND ERROR REPORTING

Structure

5.0 INTRODUCTION

The 'C' Preprocessor is not part of the compiler, but is a separate step in the compilation process. In simple words, the 'C' preprocessor is a text processor which operates on the 'C' source code before it is actually parsed by the compiler. It provides macro and conditional features very closed to the ones available with most of theexisting assemblers.

The preprocessor modifies the 'C' program source code according to special directives found in the program itself. Preprocessor directives start with a sharp sign '#' when found as the first significant character of a line.It must be the first nonblank character, and for readability, a preprocessor directive should begin in first column. The directives can be placed anywhere in a program but are most often placed at the beginning of a program, before main( ),or before the beginning of a particular function.The preprocessor expands the code written in the program according to the directives issued to it and then the expanded program is passed to the compiler.

Preprocessor directives are line based, and all the text of a directive must be placed on a single logical line. Several physical lines can be used but the last one end with the continuation characterbackslash '\'.

There are three basic kinds of directives: macro directives, conditional directives and control directives. These directives are listed in Table 1.

- The **macro** directives allow text sequences to be replaced by some other text sequences, depending on possible parameters.

- The **conditional** directives allow selective compiling of the code depending on conditions. Most of the time based on symbols defined by some macro directives.
- The **control** directives allow passing of information to the compiler in order to configure or modify its behavior.

Table 1: Categories of preprocessor directives

Directive	Description	Category
#define	Substitutes a preprocessor macro	Macro directives
#undef	Undefines a preprocessor macro	
#ifdef	Returns true if this macro is defined	Conditional directives
#ifndef	Returns true if this macro is not defined	
#if	Tests if a compile time condition is true	
#else	The alternative for #if	
#elif	#else an #if in one statement	
#endif	Ends preprocessor conditional	
#error	Prints error message on stderr	Control directives
#pragma	Issues special commands to the compiler, using a standardized method	
#include	Inserts a particular header from another file	
#line	redefines the current line number to the specified number	

ANSI 'C' defines the number of predefined macros. However, each one is presented for use inprogramming. The predefined macros should not be directly modified. The predefined macros with their description are as follows:

Macro	Description
DATE	The current date as a character literal in "MMM DD YYYY" format
TIME	The current time as a character literal in "HH:MM:SS" format
FILE	This contains the current filename as a string literal.
LINE	This contains the current line number as a decimal constant.
STDC	Defined as 1 when the compiler complies with the ANSI C standard.

Consider the program 5.1:

```
/* Program 5.1 */
#include<stdio.h>
main()
{
        printf("File :%s\n", __FILE__);
        printf("Date :%s\n", __DATE__);
        printf("Time :%s\n", __TIME__);
        printf("Line :%d\n", __LINE__);
        printf("ANSI :%d\n", __STDC__);
}
```

The output of the above program is:

```
        File : check.c
        Date :Sep 2 2012
        Time :04:50:24
        Line :10
        ANSI :1
```

5.1 OBJECTIVES

After going through this unit, you should be able to:

- Know preprocessor directives
- Define the macro directives
- Define the conditional directives and its use in the program
- Define the control directives and its use in the program.

5.2 MACRO DIRECTIVS

Macro Directive is a process of Expansion or substitution for an identifier in a program. The identifier will replace by a predefined string compared of one or more tokens.

The general format for **three**directives comes under the category of macro directives are:

#define IDENTIFER   replaced_string

#define IDENTIFIER(parameter_list)   replaced_string

#undef IDENTIFIER

The two first syntaxes allow a macro to be **defined**, and the third syntaxallows a previous definition to be **cancelled** (or **undefined**).

IDENTIFER is any word that follows the rules for a 'C' identifier, and may use lowercase or uppercase characters, but for readability reasons, most macro names are entered in uppercase only.

replaced_string represents all the characters from the first significant character immediately following **IDENTIFIER** (or the closing brace for the second syntax) up to the last character of the line. This character sequence will then replace the word **IDENTIFIER** each time it is found in the 'C' source file after the definition.

The **#undef** directive will cancel (or undoes) the previous definition of a macro. There will be no error message appears if the **#undef** directive tries to cancel a macro which is not defined previously. However, you cannot redefine a macrowhich has already been defined. In such a case, it has to be first cancelledby a ***#undef*** directive before it is redefined.

The second syntax allows more flexibility in macro substitution. You can replace with parameters. Note that the opening brace has to follow immediately the last character of the macroname, without any whitespace. Otherwise, it is interpreted as the firstsyntax and the parameter list along with the parentheses will be part ofthe replacement sequence. Each parameter is an identifier, separatedfrom the others by a comma. For example:

#define MULTIPLY(a, b) ((a)*(b))

This macro defines the word **Multiply**along with two parameters called **a** and **b**. Parameters should appear in the replacement part, and the macroshould be used in the remaining C program with a matching number of arguments.

The program 5.2 shows how the macro substitution is performed:

```
/* Program 5.2 */
#include<stdio.h>
#define MULTIPLY(a, b)   ((a)*( b))
void main()
{       int x, y, mul;
        printf("Enter two numbers:\n");
        scanf("%d %d", &x, &y);
        mul = MULTIPLY(x, y);/* Note: MULTIPLY(x, y) is replaced
        by x*y   */
        printf("The multiplication of %d and 5d is 5d\n", z);
}
```

The preprocessor is recognizing **MULTIPLY**as a valid macro, and successfully matched **a**with **x**, and **b** with **y**. The macro name and the arguments with the parentheses have been replaced by the replacementcontent of the macro **a * b** where **a** and **b** were replaced by their values **x** and **y**.

The operator **#(hash)**placed before a parameter name will turn it into a text string by enclosing it by double quotes. For example:

```
#define STRING(str) #str
```

will convert:

```
ptr = STRING(world);
```

into

```
ptr = "world";
```

This feature is useful in string concatenation.

5.3 CONDITIONAL DIRECTIVES

The **conditional** directives allow compilation of selective codes depending on conditions based on symbols defined by some macro directives. The **conditional** directives are:

#ifdef IDENTIFIER

#ifndef IDENTIFIER

#if expression

and are associated with the **ending** directives

#else

#endif

#elif expression

Any of the above conditional directive is always followed by an ending directive. A 'C'program having lines enclosed by the conditional directive and its ending directivewill be compiled or skipped depending on the result of the conditiontest.

The **conditional** directive **#ifdef** stands for "if **has** been defined". **#ifdef IDENTIFIER**is true if there is the macro **IDENTIFIER** has beenpreviously defined. For example, in the #ifdef-#endif directive pair:

```
#ifdef INCREMENT_X
        x = x + 1;
#endif
```

The statement x = x +1 will be compiled if the macro INCREMET_X has been defined previously.

The **conditional** directive# **ifndef IDENTIFIER**is true if **IDENTIFIER** is not the name of a macropreviously defined.The #ifndef stands for "if **has not** been defined". For example, in the #ifndef-#endif directive pair:

```
#ifndef SUM_B_AND_C
        Sum = b + c;
#endif
```

the statement **Sum = b +c;**will only be compiled if **SUM_B_ AND_Chas not** been defined previosuly. If **SUM_B_ AND_C has** been defined, then the statements will bedeleted.

#if expression is true if the result of expression is not zero.

The **expression** is evaluated as a constant expression, thus after all the possible macro replacements inside the expression,any word which has not been replaced by a number or an operator is replaced by the value zero before the evaluation.

An **ending** directive can also work as a conditional directive by starting a new conditional block, such as **#else** and **#elif.**

Here, are the few possible constructs:

```
#ifdef   CHECK
        printf("outline 1\n");
#endif
```

If the symbol **CHECK** has been defined previously, the line *printf...* willbe compiled properly and executed. Otherwise, it is simply skipped.

```
#if TERMINAL == 1
        initialize_console();
#else
        initialize_printer();
#endif
```

If the symbol **TERMINAL**has been previously defined to **1**, the line**initialize_console();** is compiled, and the line **initialize_printer** is skipped. If **TERMINAL**is not defined as 1, or itis not defined, the behavior of expression is just reverse (if **TERMINAL**is not defined, it is replaced by **0** andthe expression **0==1** is false).

The **#elif** directive is simply a reduction of a **#else** immediately followed by a **#if**. It avoids too complex embedding in case of multiple values:

```
#if TERM == 1
...
#elif TERM == 2
...
#elif TERM == 3
...
#else
...
#endif
```

Consider another statement:

```
#undef FILE_SIZE
#define FILE_SIZE 56
```

The above second statement will only be compiled if FILE_SIZE has not been defined previously, then FILE_SIZE has been set to 56.

5.4 CONTROL DIRECTIVES

The **control** directives allow passing of information to the compiler in order to configure or modify its behavior. The control directives are:

(a) #include

```
#include "filename"
or
#include <filename>
```

The preprocessor replaces <filename>or"filename" by the full content of the file. A filename specified between double quotes is searched first in the current directory. A filename specified between angle brackets is searched first in some predefined system directories, or user specified directories. An error is generated if the file is not found in any of the specified directories. An included file may contain other **#include** directives.

The #**include** directive can beused in two cases:

i. If we have a very large program, the code is best divided into several different file, each containing a set of related functions.

ii. Many times we need some functions or some macro definitions almost in all programs that we write.

It is common for the file which is to included to have**. h** extension. This extension stands for "**headerfile**", possibly because it contains statements which when included go to the head of a program.

(b) #error

```
#error   error_message_to_be_print
```

On seeing this directive, the compiler gives an error message whose content is the **error_message_to_be_print**. This directive is interesting to force an error if something is detected wrong in the defined symbols. For example, consider the code below:

```
#ifndef SYMBOL
        #error missing definition for SYMBOL
    #endif
```

If the **SYMBOL** is not defined, the compiler will output an error message containing the text "**missing definition for SYMBOL**", and will fail to compile the source file.

(c) #line

```
#line number "filename"
```

The above directives redefine the current line number to the specified **number**, and the **filename** to the specified name in the text string. This is mainly used by automatic code generators to allow an error to refer to the input file name and line number rather than the intermediate Csource file produced. This is almost never used by a human written program. Note that this directive modifies the value of the predefined symbol__**FILE**__.

5.5 ERROR REPORTING

Although 'C'language does not offer direct support for error reporting but as a system programming language, it gives you the facility to accesslower level contents in the form of return values. Many of the 'C' or even Unix function calls return -1 or NULL in case they found any error and sets an error code **errno** which is a global variable and indicates an error occurred during any function call. The various error codes are defined in **<error.h>**header file.

An expert 'C' programmer can check the returned values as specified in errno.h file and can take suitable action depending on the return value. It is a good practice by a developer to set **errno** to 0 at the time of initialization of the program. A value of 0 indicates that there is no error in the program.

5.5.1 The errno, perror() and strerror()

The 'C' language has two functions, namely; **perror()** and **strerror()** functions which can be used to display the text message associated with **errno**.

- The syntax of perror() function is as follows:

void perror(const char *str)

perror(s) prints str and followed by a colon, a space, and then the implementation-defined error message corresponding to the integer in **errno**.

- The **strerror()** function, which returns a pointer to the textual representation of the current errno value. The syntax of **strerror()** is as follows :

char *strerror(n);

returns pointer to implementation-defined string corresponding to error **n**.

Consider a program 5.3 that tries to open a file which does not exist. Here, both the functions are shown for demonstration, but you can use any one of the above functions for printing your errors. Another thing which is to be noted that, you should use **stderr** file stream to output all the errors.

```
/*  Porgram 5.3  */
#include <stdio.h>
#include <errno.h>
#include <string.h>
extern int errno ;        /* errno is global variable, so extern is used */
int main ()
{
        FILE *fp;
        int errnum;
        fp = fopen ("NotExist.txt", "rb");
        if (fp == NULL)
{
        errnum = errno;
        fprintf(stderr, "Value of errno: %d\n", errno);
        perror("Error printed by perror");
        fprintf(stderr, "Error opening file: %s\n", strerror( errnum ));
}
else
{
        fclose (fp);
}
        return 0;
}
```

The output of the above program is:

```
Value of errno: 2
Error printed by perror: No such file or directory
Error opening file: No such file or directory
```

Let us consider another problem when you try to divide a value by zero. As we know that the answer is undefined. But it is a common problem that programmers do not check whether a divisor is zero at the time of dividing any number, and finally it creates a runtime error.

The program 5.4 below shows this type of error and suggests a way to fix such problem.

```
/* Program 5.4 */
#include <stdio.h>
#include <stdlib.h>
main()
{
int dividend = 45;
int divisor = 0;
int quotient;
if( divisor == 0){
fprintf(stderr, "An attempt has been made to division by zero!
Exiting...\n");
exit(-1);
}
quotient = dividend / divisor;
fprintf(stderr, "Value of quotient : %d\n", quotient );
exit(0);
}
```

The function **exit(0)** indicates a **successful** termination of program, while any **non-zero value** indicates **abnormal** termination of program.

The output of the above program is:

```
An attempt has been made to division by zero! Exiting...
```

Instead of placing 0 or any another non-zero value, it is a good programming practice to exit with a value of **EXIT_SUCCESS** after a successful completion of operation. Here EXIT_SUCCESS is a macro and it is defined as 0. But in case if you think that some segment gives a runtime error then, you should exit with a status **EXIT_FAILURE** which is defined as -1. Let us consider the following program to demonstrate the use of **EXIT_SUCCESS.**

```
#include <stdio.h>
#include <stdlib.h>
main()
{
        int dividend = 45;
        int divisor = 3;
        int quotient;
        if( divisor == 0){
                fprintf(stderr, "Division by zero! Exiting...\n");
                exit(EXIT_FAILURE);
}
        quotient = dividend / divisor;
        fprintf(stderr, "Value of quotient : %d\n", quotient );
        exit(EXIT_SUCCESS);
}
```

The output of the above program is:

```
        Value of quotient:15
```

Check your progress 1

1.
```
# define dp(e) printf(#e " = %d\n",e)
main( )
{int x =3, y = 2;
dp(x/y)
}
```
What will be the output of the program?

2. What will be the output of the program?
```
#define START  0       /* Starting point of loop          */
#define ENDING 9       /* Ending point of loop            */
#define MAX(A,B) ((A)>(B)?(A):(B)) /* Max macro definition */
#define MIN(A,B) ((A)>(B)?(B):(A)) /* Min macro definition */
int main()
{int index, mn, mx;
int count = 5;
for (index = START ; index <= ENDING ; index++)  {     mx =
MAX(index, count);    mn = MIN(index, count);    printf("Max
is %d and min is %d\n", mx, mn);  }  return 0;}
```

3. What will be the output of the program?
```
#define WRONG(A) A*A*A      /* Wrong macro for cube
```

```
*/#define CUBE(A) (A)*(A)*(A)   /* Right macro for cube   */
#define SQUR(A) (A)*(A)       /* Right macro for square  */
#define ADD_WRONG(A) (A)+(A)  /* Wrong macro for addition */
#define ADD_RIGHT(A) ((A)+(A)) /* Right macro for addition */
#define START 1
#define STOP 7
int main()
{ int i, offset;
offset = 5;
 for (i = START ; i <= STOP ; i++)
 {
printf("The square of %3d is %4d, and its cube is %6d \n", i+offset,
SQUR(i+offset), CUBE(i+offset));
 printf("The wrong of %3d is %6d \n", i+offset, WRONG(i+offset));
}
 printf("\nNow try the addition macro's \n");
for (i = START ; i <= STOP ; i++) {
printf("Wrong addition macro = %6d, and right = %6d \n",
5*ADD_WRONG(i), 5*ADD_RIGHT(i));
}
 return 0;}
```

What will be the output of the program?

```
#define OPTION_1     /* This defines the preprocessor control
*/#ifdef OPTION_1   int count_1 = 17;   /* This exists only if
OPTION_1 is defined */
#endifint main()
{int index;
for (index = 0 ; index < 6 ; index++) .
 {
printf("In the loop, index = %d", index);
#ifdef OPTION_1
printf(" count_1 = %d", count_1); /* This may be printed    */
#endif
printf("\n");
 }  return 0;
}
#undef OPTION_1
```

What will be the output of the program?

```
#define OPTION_1
```

```
 #ifndef OPTION_1
        int count_1 = 17;
#endifint main()
{int index;
#ifndef PRINT_DATA
printf("No results will be printed with this version of the program
IFNDEF.C\n");
#endif  for (index = 0 ; index < 6 ; index++)
{
#ifdef PRINT_DATA
printf("In the loop, index = %d", index);
#ifndef OPTION_1
 printf(" count_1 = %d", count_1);
 #endif    printf("\n");
#endif
}  return 0;
}
```

5.6 SUMMARY

The preprocessor directives are powerful tool for manipulating text files. While control directives, macro directives and conditional compilation are its most popular features. Being'C'a system programming language, it provides you access at lower level in the form of return values. In case of any error, compiler sets an error code **errno** which is global variable and indicates an error occurred during any function call. You can find various error codes defined in **<error.h>**header file.