

PROGRAMMING IN C

CONTENT AT A GLANCE

MODULE 1

Unit 1 : Basics of Programming

Unit 2 : Fundamentals

Unit 3 : C Operators

MODULE 2

unit 1 : Input Output Statements

unit 2 : Control Structures

unit 3 : Program Looping

MODULE 3

Unit 1 : Functions

Unit 2 : Variables and Storage Classes

MODULE 4

Unit 1 : Arrays

Unit 2 . Strings

Module 5

Unit 1 : Pointers

Unit 2 : Macros and Preprocessors

Module 6

Unit 1 : Structures

Unit 2 : Unions

TABLE OF CONTENTS

MODULE 1

UNIT 1: Basics of Programming

- 1.1 Introduction
- 1.2 Problem Solving Techniques
- 1.3 Algorithm
 - 1.3.1 Properties of an Algorithm
 - 1.3.2 Basic Statements used and Examples
- 1.4 Flow charts
 - 1.4.1 Types of Flowcharts
 - 1.4.2 Symbols used in Flowcharts
 - 1.4.3 Writing Expressions in Computer Language
 - 1.4.4 Examples
- 1.5 Programming by using Simple Flowcharts
 - 1.5.1 Adding first N Numbers
 - 1.5.2 Fahrenheit Scale Convert into Celsius Scale
 - 1.5.3 Area and perimeter of a Triangle
- 1.6 Questions
- 1.7 Programing Exercise.

Unit 2 : C Fundamentals

- 2.1 What is C ?
- 2.2 Program Structure in C
- 2.3 Executing a C program
- 2.4 C character set
- 2.5 Identifiers and Keywords
 - 2.5.1 Keywords
- 2.6 Basic Data Types
- 2.7 Modifiers to Basic Data Types
- 2.8 Constants
 - 2.8.1 Integer constants
 - 2.8.2 Decimal Integer Constants
 - 2.8.3 Octal Integer Constants
 - 2.8.4 Hexadecimal Integer Constants
 - 2.8.5 Unsigned and Long Integer Constants
 - 2.8.6 Floating Point Constants
 - 2.8.7 Character Constants
 - 2.8.8 String Constants
- 2.9 Variables
 - 2.9.1 Variable Declarations
 - 2.9.2 Assigning Values to Variables
 - 2.9.3 Declaring a variable as constant
- 2.10 Questions
- 2.11 Programing Exercise.

Unit 3 : Operators In C

- 3.1 Introduction
- 3.2 Arithmetic Operators
- 3.3 Relational Operators
- 3.4 Logical Operators
- 3.5 Conditional Operator
- 3.6 The size of Operator
- 3.7 Precedence of Operators
- 3.8 Questions
- 3.9 Programing Exercise

MODULE 2

unit 1 : Input Output Statements

- 1.1 Introduction
- 1.2 Formatted I / O Functions
- 1.3 The printf() Function
 - 1.3.1 Application of printf() Function
 - 1.3.2 Escape Sequence in printf() Function
 - 1.3.3 Minimum Field Width Specifier (MFWS)
 - 1.3.4 The Precision Specifier
- 1.4 The scanf() Function
- 1.5 Unformatted I / O Function
- 1.6 Character Input/Output(I/O)
- 1.7 String I / O
- 1.8 Questions
- 1.9 Programming exercises

unit 2: Control Structures

- 2.1 Introduction
- 2.2 The if – else statement
- 2.3 Nested -ifs
 - 2.3.1 The if -else - if Ladder
- 2.4 Switch Statement
 - 2.4.1 Rules of Switch Statement
 - 2.4.2 Difference Between if Statement and Switch Statement
- 2.5 Questions
- 2.6 Programming Exercise.

unit 3:Program Looping

- 3.1 Loops
- 3.2 The *for* loop
 - 3.2.1 The for Loop Variations
 - 3.2.2 The Infinite Loop
 - 3.2.3 The Comma Operator in The for Loop
 - 3.2.3 Declaring Variables Inside The for Loop
 - 3.2.4 The Nested for Loop
- 3.3 While Loop
 - 3.3.1The Infinite while Loop
- 3.4 Do-while Loop
 - 3.4.1 Limitation of Do-while Loop
- 3.5 Break Statement
 - 3.5.1 Break Statement Within Loops
 - 3.5.2 Can break Statement be Used Inside if Construct
- 3.6 Continue Statement
- 3.7 Goto and Labels
- 3.8 Questions
- 3.9 Programming Exercise

MODULE 3

Unit 1 : Functions

- 1.1 Introduction
- 1.2 What is a Function?
 - 1.2.1 Functions are Used in C for the Following Reasons
- 1.3 Structure of Function
 - 1.3.1 Function Definition
 - 1.3.2 Function Prototypes
 - 1.3.3 Function Invocation
- 1.4 Types of Arguments
- 1.5 Types of Functions
 - 1.5.1 A Function With No Arguments And No Return Value
 - 1.5.2 A Function With No Arguments and Returns a Value
 - 1.5.3 A Function With Arguments and Returns No Value
 - 1.5.4 A Function With Arguments and Returning a Value
- 1.6 Questions
- 1.7 Programing Exercise

Unit 2 : Variables and Storage Classes

- 2.1 Local variables
- 2.2 Global Variables
- 2.3 Call by Value and Call by Reference
 - 2.3.1 Call by Value
 - 2.3.2 Call by Reference
- 2.4 Recursion
- 2.5 Storage Class
 - 2.5.1 Local or Automatic Variables
 - 2.5.2 Global or External Variables
 - 2.5.3 Static Variables
 - 2.5.4 Register Variables
- 2.6 Questions
- 2.7 Programming Exercise

MODULE 4

UNIT 1: Arrays

- 1.1 Introduction
- 1.2 Defining an Array
 - 1.2.1 Declaring Single Dimensional Array
 - 1.2.2 Array Index
- 1.3 Initialization of One Dimensional Array
 - 1.3.1 Compile Time Initialization
 - 1.3.2 Run Time Initialization
- 1.4 Entering Data Into The Array
- 1.5 Reading Data From Array
- 1.6 The Size of an Array
- 1.7 Multidimensional Arrays
 - 1.7.1 Declaration of Two-dimensional Array
 - 1.7.2 Two-Dimensional Array Initialization
- 1.8 Processing a Two Dimensional Array
 - 1.8.1 Entering Data into Two Dimensional Array
 - 1.8.2 Printing data of Two Dimensional Arrays
- 1.9 Passing Arrays as Arguments
- 1.10 Questions
- 1.11 Programing Exercise

UNIT 2: Strings

- 2.1 Introduction
- 2.2 Declaring a string

- 2.3 Initializing a String
 - 2.3.1 Type 1
 - 2.3.2 Type 2
- 2.4 Initialization After Declaration
- 2.5 String Constants Versus Character Constants
- 2.6 Input data to string
 - 2.6.1 Reading Strings Using scanf ()
 - 2.6.2 Reading Strings using gets()
- 2.7 Print strings
 - 2.7.1 Printing strings using printf()
 - 2.7.2 Printing strings using puts()
- 2.8 Built-in String Functions
- 2.9 C Character Functions
- 2.10 Questions
- 2.11 Programing Exercise

MODULE 5

Unit 1 : POINTERS

- 1.1 Introduction
- 1.2 How variables are stored in memory?
- 1.3 What is a Pointer?
- 1.4 Pointer Declaration
 - 1.4.1 Pointer Declaration Styles
 - 1.4.2 Multiple Declarations
- 1.5 Pointer Initialization
 - 1.5.1 The Address of Variables
 - 1.5.2 Null Pointers
 - 1.5.3 Understanding pointers
- 1.6 Pointer Expressions
- 1.7 Pointer Arithmetic
 - 1.7.1 Pointer increment and decrement
 - 1.7.2 Pointer Addition and Subtraction
 - 1.7.3 Pointer Multiplication and Division
- 1.8 Pointers and Function
 - 1.8.1 Call by Reference
 - 1.8.2 Call by Value
- 1.9 Pointers and Arrays
 - 1.9.1 Pointers and One Dimensional Array
 - 1.9.2 Pointers and Strings
- 1.10 Questions
- 1.11 Programing Exercise

Unit 2 : Macros and Preprocessors

- 2.1 Introduction
- 2.2 Preprocessor Directives
- 2.3 Macro Substitution Directives
 - 2.3.1 Simple Macro Substitution
 - 2.3.2 Macro Inside the Quotes
 - 2.3.3 Macros With Parameters
 - 2.3.4 Undefining a Macro
- 2.4 File Inclusion
- 2.5 Questions
- 2.6 Programing Exercise

MODULE 6

Unit 1: STRUCTURES

- 1.1 Introduction
- 1.2 Defining a Structure
- 1.3 Declaring Structure Variables
- 1.4 Accessing Structure Members
 - 1.4.1 Assigning Values to the Members
 - 1.4.2 Structure Initialization
- 1.5 Questions
- 1.6 Programing Exercise

unit 2:Unions

- 2.1 Introduction
- 2.2 Declaration of Union
- 2.3 Working With Unions
- 2.4 Difference between Structures and Unions
- 2.5 Questions
- 2.6 Programing Exercise

MODULE 1

Unit1 : Basics of Programming

1.1 Introduction

Computer programming is a set of instructions through which one tells the computer to do the desired task. This set of instructions written in human readable computer language is called Source Code. Every program has two parts namely code and data. There are two models of programming namely Structured Programming and Object Oriented Programming. In Structured Programming codes are executed one after one in a serial fashion. Example for this is 'C' Language. In object oriented programming, data is accessed through objects. There is no single flow. Here objects freely interact with one another by passing messages.

1.2 Problem Solving Techniques

To develop the solution for the given problem, the following programming techniques are used.

Problem solving techniques

i) Algorithm

It is a set of logical procedure steps to solve the problem.

ii) Flow Charts

It is a diagrammatic representation of the sequence of operation for a given problem.

iii) Pseudo codes

These are the instructions written in ordinary English using mathematical and logical symbols to display the program logic.

iv) Decision Tables

A decision table consists of many independent conditions with several actions, written in table format to solve the given problem.

1.3 Algorithm

The word 'Algorithm' is the name of one Persian author meaning rules of restoration and reduction. Once the problem is analyzed, its solution is broken into a number of sample steps. A problem in a finite sequence is called an algorithm.

1.3.1 Properties of an Algorithm

- i. **Finiteness:** An algorithm must always terminate after a finite number of steps.
- ii. **Definiteness:** Each step must be clearly defined that actions carried out must be unambiguous.
- iii. **Input:** Input should be provided at the beginning of algorithm.
- iv. **Output:** Algorithm must produce one or more output.
- v. **Effectiveness:** All the operations defined must be sufficiently basic that they can be done exactly in finite length of time manually.

1.3.2 Basic Statements Used and Examples

- i. Algorithm always begins with the word 'Start' and ends with the word 'Stop'
- ii. Step wise solution is written in distinguished steps. This is as shown in example 1.1

```

Start
Step 1:
Step 2:
Step 3:
.
.
.
Step n:
End
    
```

Example 1.1 Structure of an Algorithm

- iii. **Input Statement:** Algorithm takes one or more inputs to process. The statements used to indicate the input is Read a or Input b. This is illustrated in example 1.2

Let a , b be the names of the Input
 Input a or Read a
 Input b or Read b
 Where a and b are variable names.

Example 1.2 Format of Input Statement in Algorithm

- iv. **Output Statements:** Algorithm produces one or more outputs. The statement used to show the output is output a or print b. This is illustrated in example 1.3

Syntax: Output variable name
 Print variable name

For example output a or print a
 output b or print b
 where a and b are variable names.

Example 1.3 Format of output Statement in Algorithm

- v. **Assignment Statements:** Processing can be done using the assignment statement.

i.e. L.H.S = R.H.S
 On the L.H.S is a variable.

While on the R.H.S is a variable or a constant or an expression. The value of the variable, constant or the expression on the R.H.S is assigned in L.H.S. The L.H.S and R.H.S should be of the same type. Here ' = ' is called assignment operator. This is illustrated in example 1.4

Let the variables be x, y. The product be z this can be represented by as

Read x, y
 $Z = x * y$

Example 1.4 Format of Assignment Statement in Algorithm

- vi. Order in which the steps of an algorithm are executed is divided in to 3 types namely

- i) Sequential Order
- ii) Conditional Order
- iii) Iterative Order

i) Sequential Order

Each step is performed in serial fashion I.e. in a step by step procedure this is illustrated in example 1.5

Task : Write an algorithm to add two numbers.

Step 1 : Start
 Step 2 : Read a
 Step 3 : Read b
 Step 4 : Add a , b
 Step 5 : Store in d
 Step 6 : Print d
 Step 7 : End

Example 1.5 Format of Sequential order Algorithm

ii) Conditional Order

Based on fact that the given condition is met or not the algorithm selects the next step to do. If statements are used when decision has to be made. Different format of if statements are available they are

a) Syntax :

if (condition)
 Then {set of statements S1}

Here condition means Boolean expressions which evaluates to TRUE or FALSE. If condition is TRUE then the statements S1 is evaluated. If false S1 is not evaluated Programme skips that section. This is illustrated in example 1.6

Task : Write an algorithm to check equality of numbers.

Step 1 : Start
 Step 2 : Read a, b
 Step 3 : if a = b, print numbers are equal to each other
 Step 4 : End

Example 1.6 Format of ' if ' Conditional order Algorithm

b) Syntax – if else (condition)

if (condition)
 Then {set of statements S1}
else
 Then {set of statements S2}

Here if condition evaluates to true then S1 is executed otherwise else statements are executed. This is illustrated in example 1.7

Task : Write an algorithm to print the grade.

```
Step 1 : Start
Step 2 : Read marks
Step 3 : Is marks greater than 60 ?
Step 4 : if step 3 is TRUE
    print 'GRADE A'
Step 5 : Other wise
    print 'GRADE B'
Step 6 : End
```

Example 1.7 Format of Conditional order Algorithm

c) Syntax – Nested if else (condition)

```
If (condition 1)
    Then S1
Else
    If (condition 2)
        Then S2
    Else
        Then S3
```

Here if and else condition is in a nested fashion this is more suited for the programs have been multiple conditions. This is illustrated in example 1.8

Task : Write an algorithm to find grades of the marks.

```
Step 1 : Start
Step 2 : Read marks
Step 3 : Is marks greater than 60 ?
Step 4 : if step 3 is TRUE
    print 'GRADE A'
Step 5 : else if marks greater than 50 less than 60
    print 'GRADE B'
Step 6 : else print 'GRADE C'
Step 7 : End
```

Example 1.8 Format of nested if else Conditional order Algorithm.

iii) Iterative Order

Here algorithm repeats the finite number of steps over and over till the condition is not meet. Iterative operation is also called as looping operation. This is illustrated in example 1.9

Example 1.9 : Add 'n' natural numbers till the sum is 5.

```
Step 1 : Start
Step 2 : set count to 0
Step 3 : add 1 to count
Step 4 : if count is less than 5,
```

Repeat steps 3 & 4
Step 5 : otherwise print count
Step 6 : End

Example 1.9 Format of iterative order Algorithm.

1.4 Flow Charts

Algorithm for large problems becomes complex and there by difficult to write the code. Problem analysts found '**Flow charts**' an easier way to solve the problem. Here each step is represented by a symbol and also contains a short description of the process steps within the symbol. Flow charts are linked by arrows. The other names of flow chart are flow diagram, process chart, and business flow diagram etc., most often it is called by name flow chart only. Flow charts can be used for following aspects

- i. Define and analyze
- ii. Build step by step picture of a process
- iii. To find the areas of improvement in the process

Advantages

- i. Communication – it is better way of communicating the logic of a program solution.
- ii. Effect analysis – with the help of flow charts problem can be analyze in a effective way
- iii. Effective coding – flow charts acts as a blue print during the system analysis
- iv. Proper debugging – flow chart helps in debugging process.

Limitations

- i. Alterations and modifications: If alterations are required, the flowchart may require re-drawing completely.

1.4.1 Types of Flowcharts

Flow charts can be broadly classified in to two types

i) system flow charts

These are used by system analyst to describe the data flow and operation in a data processing cycle. System flow chart defines the broad processing in organizations showing the origin of data filing structure, processing to be performed and output to be generated.

ii) Program flow chats

Program flow charts are used by programmers. It is used to describe the sequence of operations and decisions for a particular problem. Generally

to solve any of the programs belonging to COBOL, C, C++, Java. . etc., program flow charts are used.

1.4.2 Symbols Used in Flowcharts

The flow chart being symbolic representation standard symbols is used for each specific operation. These symbols are used to represent the sequence of operations and flow of data and documents required for programming. Flow should be from top to bottom. The most commonly used symbols are shown in the figure 1.1

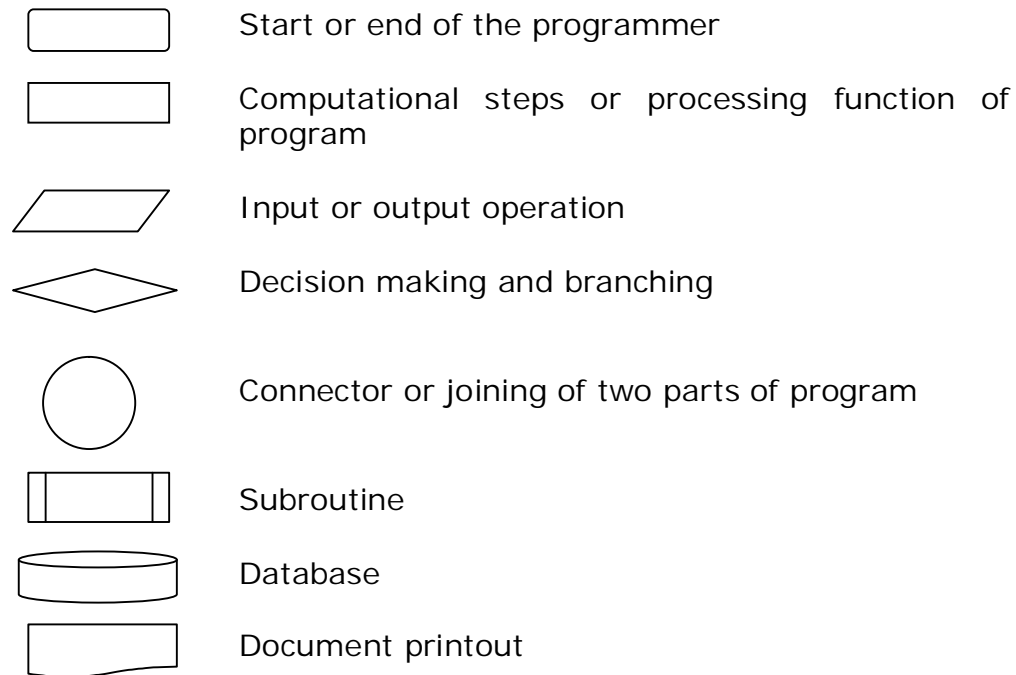
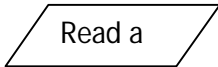


Figure 1.1 Flowchart symbols

1.4.3 Writing Expressions in Computer Language

- i. **To read the input ;**
 - i. The following symbol is used to read one or more inputs.


 - ii. Here 'a' is the variable where the value of Input is stored.
- ii. **To produce output :**
 - i. The following symbol is used to print one or more outputs.

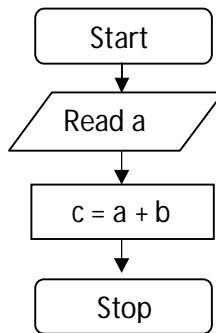


ii. Here 'a' is the variable where the value of Output is stored.

iii. Assignment operator

The result of arithmetic operation is stored in a variable. This is represented as below. This is illustrated in example 1.10

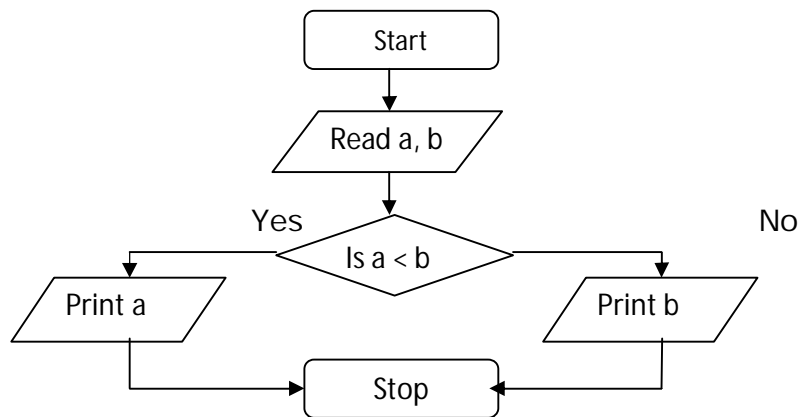
1. Let a and b be inputs
2. Operation \longrightarrow addition
3. Result is stored in variable c



Example 1.10 Assignment operator.

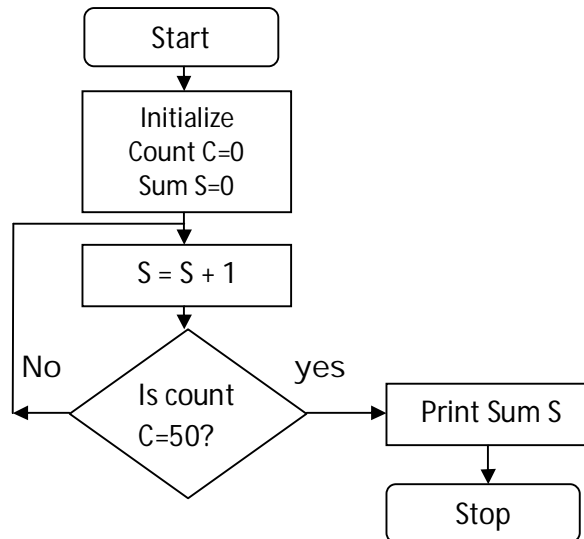
1.4.4 Examples

- i) Draw a flowchart to find greater of 2 numbers



Example 1.11 Flowchart of greatest of 2 numbers

- ii) Write a flowchart to add natural numbers from 1 to 50

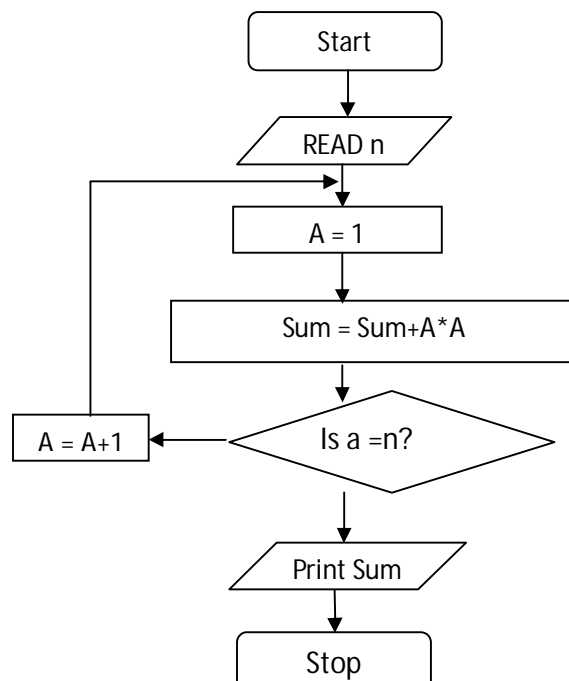


Example 1.12 Flowchart to find sum of natural numbers.

1. 5 Programming by Using Simple Flowcharts

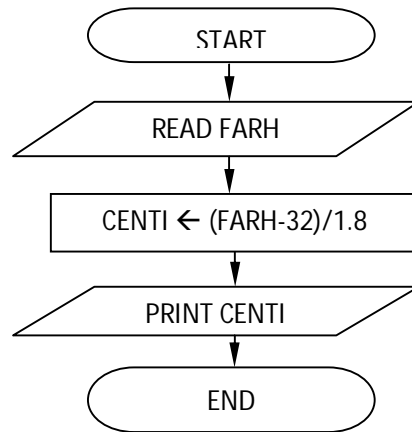
To understand the concept of programming using flowcharts, consider the below examples.

1.5.1 Adding First N Numbers



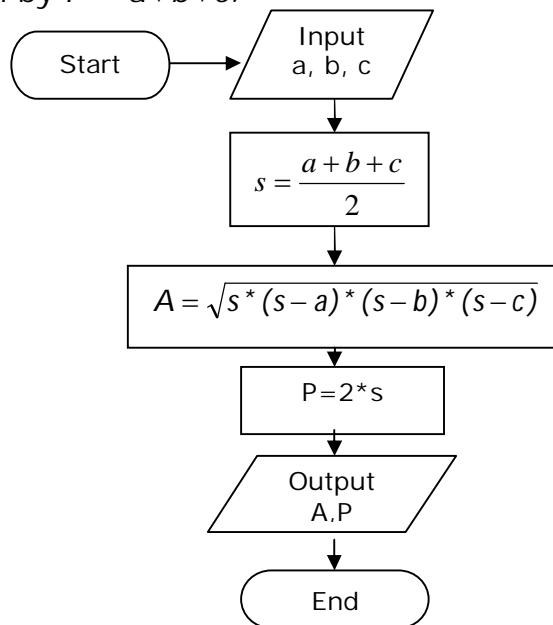
1.5.2 Fahrenheit Scale Convert into Celsius Scale

Solution: Formula: - $C = (F-32)/1.8$ where F is Fahrenheit value and C is Celsius value.



1.5.3 Area and Perimeter of a Triangle

Solution: Formula for calculating the area of a triangle with the length of the three sides as a, b, c is given by $A = \sqrt{s * (s-a) * (s-b) * (s-c)}$, where s is the semi-perimeter of the triangle, $s = \frac{a+b+c}{2}$. The perimeter of the triangle is given by $P = a+b+c$.



Step 1: [Input the three sides of triangles]

Input A,B,C

Step 2: [Check the validity of given inputs and compute the area if the inputs valid]

IF ((A+B)>C OR (B+C)>A OR(C+A)>B) then

$S \leftarrow (A+B+C)/2$

```

        AREA = SQRT(S*(S-A)*(S-B)*(S-C))
        Write "Area of triangle is ",AREA
    ELSE
        Write "Given inputs are invalid"

```

Step 3: [End of algorithm]
End.

1.6 Questions

- 1.Explain problem solving techniques in brief
- 2.What is an algorithm and how it is presented?
- 3.Define flowchart. State its advantages and limitations.
- 4.Explain various symbols of flowchart.

1.7 Programing Exercise

- 1.Write an algorithm and draw a flowchart to add the following sequence
 - a. 1, 3, 5, 7, 9 . . . N (series of odd numbers)
 - b. 2, 4, 6, 8, 10 . . . N (series of even numbers)
- 2.Draw flowchart and algorithm to find gross salary.
- 3.Write an algorithm and flow chart to determine whether the given integer array is palindrome or not
4. Write a flow chart to find the sales price of a product using the formula sales price = cost price of a product + profit of 10%

MODULE 1

Unit 2 : C Fundamentals

2.1 What is C ?

C is a programming language developed at AT & T laboratories of USA in 1972 by Dennis Ritchie in 1970. It was initially implemented on the system

that used UNIX operating system. C is a midlevel computer language. C is a nice blend of high level languages and low level assembly language. C code is portable. C language position in the areas of programming languages is shown in Table 2.1

| | |
|---------------------|--|
| High Level Language | ADA PASCAL COBOL FORTRAN BASIC |
| Middle Level | JAVA C++ C |
| Low Level | Assembly Language |

Table 2.1 Arena of programming Languages

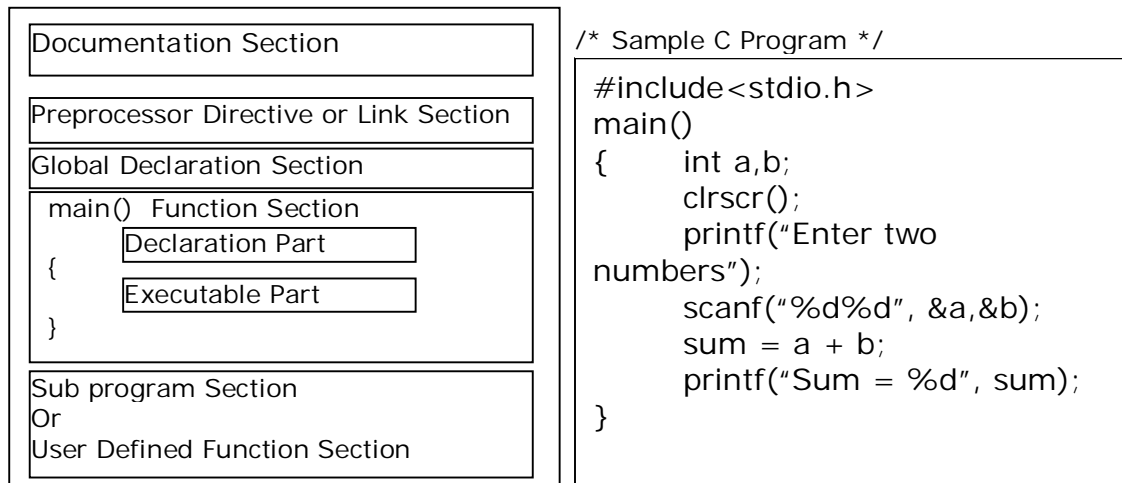
Note that *Portability* means that it is easy to adapt software written for one type of OS to run on another type.

Important Features of C Language

- i. C is a system programming language which provides flexibility for writing compilers, operating systems, etc.
- ii. It can also be used for writing the application programs for scientific, engineering and business applications.
- iii. C is famous for its portability, meaning that program written in C for one computer can be easily loaded to another computer with little or no changes.
- iv. C supports variety of data types like integers, float point numbers, characters, etc.
- v. C is a procedure oriented language which is most suited for structured programming practice.
- vi. It provides a rich set of built in functions

2.2 Program Structure in C

To accomplish the given task programs are written and it is executed in Turbo C/C++ compiler. The structure of the program is given below.



- i. The Documentation Section: It consists of a set of comment lines giving the name of the program, the author and other details which the programmer would like to use later. The comments are enclosed in a C program using /* and */ .
- ii. The Preprocessor directive or Link section: It provides instruction to the compiler to link some functions or do some processing prior to the execution of the program. It is also used to define symbolic constants of the program.
 - Header file used by the example is a standard input/output file (stdio.h).
 - programs must contain an #include line for each header file
 - There are two punctuation forms for head file.
 1. #include <stdio.h>
 2. #include "stdio.h"
- iii. Global Declaration Section : There are some variables that are used in more than one function. Such variables are called global variables and are declared in this section that is outside of all other functions.
- iv. The main() function section : Every C program must have one main() function. This section contains two parts, declaration part and executable part. The declaration part declares all the variables used in the executable part. There is at least one statement in the executable part. These two parts must appear between the opening and closing braces. The program execution begins at the opening brace and ends at the closing brace. The closing brace of the main function is the logical end of the program. All statements in the declaration part and executable parts must end with a semicolon.

- v. Sub program Section : It contains all the user defined functions that are called in the main() function. User defined functions are generally placed immediately after the main function, although they may appear in any order.
- vi. All sections except the main() function may be absent when they are not required in any C program.
- vii. Note that each of these topics are explained in detail in the following chapters of the book.

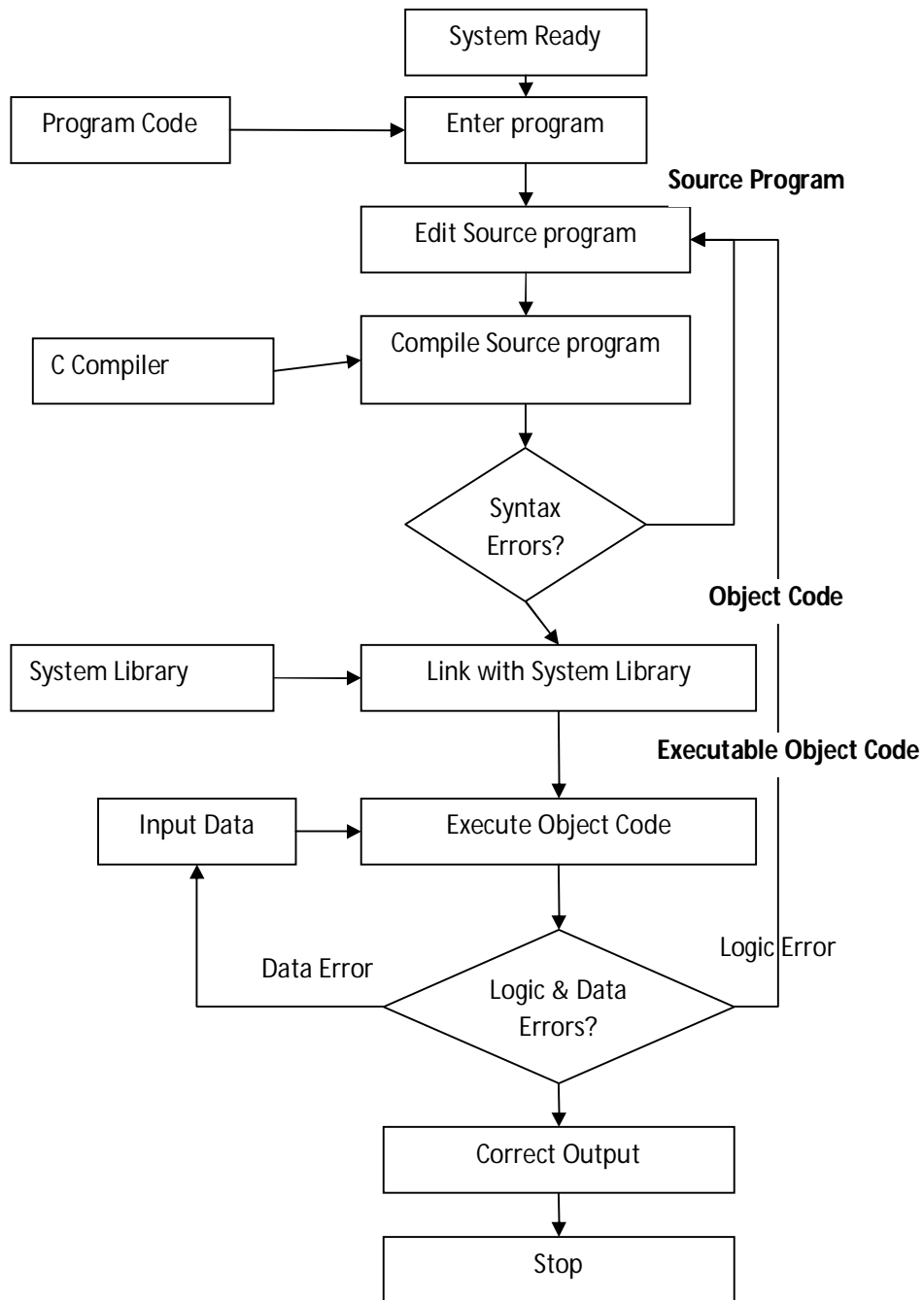
2.3 Executing a C program

Executing a C program involves a series of following steps.

- i. Creating a program
- ii. Compiling the program
- iii. Linking the program with functions that are needed from the C library.
- iv. Executing the program

It is represented in the below flowchart.

PROGRAMMING IN C



In MS DOS :

For Compiling press Alt-F9.

For Compiling & Running the program press Ctr-F9.

For viewing the results press Alt-F5.

Important points to remember:

1. Every C program requires a main() function. Use of more than one main() is illegal. The place of main() is where the program execution begins.
2. The Execution of the function begins at the opening brace and ends at the closing brace.
3. C programs are written in lowercase letters. However uppercase letters may be used for symbolic names and constants.
4. All the words in a program line must be separated from each other by at least one space or a tab, or a punctuation mark.
5. Every statement must end with a semicolon.
6. All variables must be declared for their type before they are used in the program.
7. Compiler directives such as define and include are special instructions to the compiler, so they do not end with a semicolon.
8. When braces are used in the program make sure that the opening brace has corresponding ending brace.
9. C is a free form language and therefore proper form of indentation of various sections would improve the legibility of the program.

2.4 C character set

Every language has its own character set. The character set of the C language consists of basic symbols of the language. A character indicates any English alphabet, digit or special symbol including arithmetic operators. The C language character set includes

1. Letter, Uppercase A Z, Lower case a....z
2. Digits, Decimal digits 0....9.
3. Special Characters, such as comma, period. semicolon; colon: question mark?, apostrophe' quotation mark " Exclamation mark ! vertical bar | slash / backslash \ tilde ~ underscore _ dollar \$ percent % hash # ampersand & caret ^ asterisk * minus – plus + <, >, (,), [,], {, }
4. White spaces such as blank space, horizontal tab, carriage return, new line and form feed.

2.5 Identifiers and Keywords

Identifiers are the names given by user to various program elements such as variables, functions and arrays. The rules for identifiers are given below.

- i. Letters, digits and underscore can be used.
- ii. Must not start with a digit.
- iii. Avoid using underscore at the start of identifier
- iv. Identifier cannot be same as *reserved word* (Key Word) or C library function names

- v. Identifiers are Case sensitive. For example india is different from India.

Examples of Valid and Invalid Identifier are given in example2.1

Valid Identifiers:

- 1) X1x
- 2) Count_2
- 3) Num1

Invalid Identifiers:

- I) 5thstandard : first character must be a letter
- II) "Sam" : illegal character (" ")
- III) Emp-no : illegal character (-)
- IV) Reg no : illegal character (blank space)

Example 2.1: Valid and Invalid Identifiers

Identifiers can be arbitrarily long. Some C compilers recognize only first 8 characters. As a rule an identifier should contain enough characters so that its meaning while reading is apparent. On the other side an excessive number of characters should be avoided.

2.5.1 Keywords

Key words or Reserve words of the C language are the words whose meaning is already defined and explained to the C language compiler. Therefore Reserve words can not be used as identifiers or variable names. They should only be used to carry the pre-defined meaning. For example int is a reserve word. It indicates the data type of the variable as integer. Therefore it is reserved to carry the specific meaning. Any attempt to use it other than the intended purpose will generate a compile time error. C language has 32 keywords. Following are some of them

| | | |
|----------------|----------|----------|
| auto | extern | sizeof |
| break | floatn | static |
| case | for | struct |
| Char | got0 | switch |
| const continue | if | typedef |
| | int | union |
| Default | long | unsigned |
| do | register | void |
| double | return | volatile |
| else | short | while |
| enum | signed | |

Some compilers may also include some or all of the following keywords.

| | | |
|-------|---------|--------|
| ada | far | near |
| asm | fortran | pascal |
| entry | huge | |

2.6 Basic Data Types

C supports five fundamental data types: Character, Integer, Floating-Point, Double floating-Point, and valueless. These are denoted as char, int, float double, void, respectively, '*void*' is typically used to declare as function as returning null value. This issue is discussed in subsequent chapters. The table 2.2 given details about fundamental data types.

| <u>Data Type</u> | <u>Description</u> | <u>Typical memory requirements</u> |
|------------------|---|---|
| int | integer quantity | 2 bytes or one word (varies from one compiler to another) |
| char | single character | 1 byte |
| float | floating-point number (i.e., a number containing a decimal point and/or an exponent) | 1 word (4 bytes) |
| double | double-precision floating-point number (i.e., more significant figures, and an exponent which may be larger in magnitude) | 2 words (8 bytes) |

Table 2.2 Fundamental data types

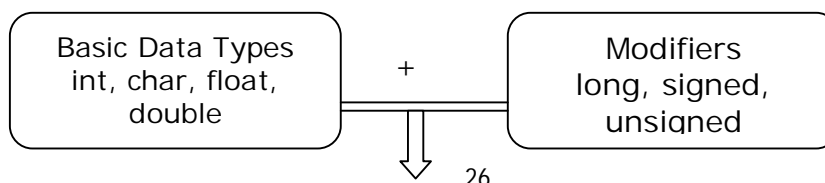
2.7 Modifiers to Basic Data Types

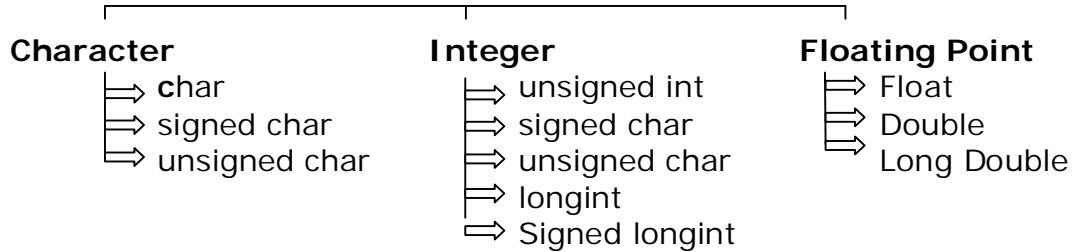
Modifiers are used to alter the meaning of basic data types to fit various needs. Except the type void, all other data types can have various modifiers preceding them.

List of modifiers used in C are:

- i) Signed
- ii) Unsigned
- iii) Long
- iv) Short

Table 2.3 shows the list of modifiers that can be used with each basic data type.





| Data Types | Modifiers | Memory Requirement | |
|----------------|------------------------|--------------------|----------------------------|
| | | Bits | Byte |
| Character | i) Char | 8 | 1 |
| | ii) signed char | 8 | 1 |
| | iii) unsigned char | 8 | 1 |
| Integer | i) int | 16 | 2 |
| | ii) signed int | 16 | 2 |
| | iii) unsigned int | 16 | 2 |
| | iv) short int | 16 | 2 |
| | v) long int | 32 | 4 |
| | vi) signed long int | 32 | 4 |
| | vii) unsigned long int | 32 | 4 |
| Floating point | i) float | 32 | 4(6 digits of precision) |
| | ii) double | 64 | 8(10 digits of precision) |
| | iii) long double | 128 | 16(10 digits of precision) |

Table 2.3 Modifiers and their memory requirements

2.8 Constants

There are four basic types of constants in C. They are integer constants, floating-point constants, character constants and string constants. Integer and floating point constants represent numbers. They are also called *numeric- type constants*.

2.8.1 Integer constants

An integer constant is an integer valued number. It consists of sequence of digits. Integer constants are divided into three different number systems. They are decimal constants (base 10), Octal constants (base 8) and hexadecimal constants (base 16). All the integer constants have to follow the set of rules given below.

- (i) Constant must have at least one digit.
- (ii) Must not have decimal point.
- (iii) Constant can be preceded by minus (-) sign, to indicate negative constants.
- (iv) Positive integer constants can be preceded by either positive sign or no sign.
- (v) Commas and blank spaces can't be included with in the constant.
- (vi) The value of constant must not exceed specified minimum and maximum bound. Generally for 16bit computer, the integer constant range is -32768 to 32767. For 32 bit computer the range is much larger.

Now let us consider each type of integer constants individually.

2.8.2 Decimal Integer Constants

A decimal integer constant consists of any combination of digits taken from the set 0 to 9. If the constant consists of two or more digits, the first digit must be other than 0. In the example 2.2 certain valid and invalid decimal integer constants are shown.

Valid decimal integer constants:

0 1 743 32767 8888

Invalid decimal integer constants

| | |
|----------|-----------------------------------|
| 080 | First digit cannot be zero |
| 50.3 | Illegal character (.) |
| 12,542 | Illegal character (,) |
| 25 35 60 | Illegal character (Blank Space) |

Example 2.2 Valid and invalid decimal integer constants

2.8.3 Octal Integer Constants

An octal integer constant consists of combination of digits taken from the set 0 through 7. However the first digit must be 0, in order to identify the constant as an octal number. In the example 2.3 some valid and invalid octal integer constants are shown.

Valid octal integer constants

00 06 0753 0663

Invalid octal integer constants

| | |
|---------|--------------------------|
| 543 | Does not begin with zero |
| 07865 | Illegal character (8) |
| 06, 593 | Illegal character (,) |
| 06.512 | Illegal character (.) |

Example 2.3 Valid and invalid octal integer constants

2.8.4 Hexadecimal Integer Constants

A hexadecimal integer is identified by `0x` or `0X`. Hexadecimal integer constant consists of digits taken from the set of 0 to 9 and letters taken from the set from A to F (either upper or lower case). The letter a to f or A to F represent the decimal values from 10 to 15 respectively i.e., a=10, b=11, c=12, d=13, e=14 and f=15. In the example 2.4 some valid and invalid hexadecimal integer constants are shown.

Valid hexadecimal integer constants:

`0x0` `0x1a` `0x1BEC` `0x7FFF`

Invalid hexadecimal integer constants

`0x16.8b` Illegal character (.)

`563c` Does not begin with 0 or 0x

`0x7bcg` Illegal character (g)

`0x12.53` Illegal character (.)

Example 2.4 Valid and invalid hexadecimal integer constant.

For most of the PC's the typical maximum value of decimal integer constant is 32767. This is equivalent to 77777 in octal or 7FFF in Hexadecimal.

2.8.5 Unsigned and Long Integer Constants

Unsigned integer constant can be identified by assigning the letter U at the end of the constant. Unsigned integer constant magnitude may exceed the magnitude of integer constant by factor of two. Long integer constants exceed the magnitude of ordinary integer constant. It can store up to four bites. Long integer constant cab be identified by appending L to l (upper or lower case) at the end of constant. An unsigned long integer is specified by UL or ul (upper or lower case) at the end of constant. Several unsigned and long integer constants are shown in the example 2.5

| Constant | Number system |
|------------|------------------------------|
| 50000u | unsigned decimal |
| 0598671243 | decimal (long) |
| 0123456789 | decimal (unsigned long) |
| 0123456 | octal (long) |
| 0X50000U | hexa decimal (unsigned) |
| 0XFFFFFUL | hexa decimal (unsigned long) |

Example 2.5 long, unsigned long integer constants.

2.8.6 Floating Point Constants

These constants are also called as 'real constants'. The real constant can be written in two forms namely factorial form and exponential form. If the value of the constant is either too small or too large the exponential form is used. The interpretation of a floating-point number in exponential form is same as scientific notation, except that base 10 is replaced by the letter E (or e). Thus the number 1.5×10^{-3} can be written as $1.5 E^{-3}$ or $1.5 e^{-3}$. This is equivalent to $0.15e-2$ or $15e-4$. Etc.,

In the exponential form the part of the constant before 'e' is called exponent. This is shown in example 2.6

$$\begin{array}{ccc} \underline{3.5} & e & \underline{+ 5} \\ \text{Mantissa} & & \text{Exponent} \end{array}$$

Example 2.6 Exponential form of real constant.

The rules for constructing real constants are as follows.

- (i) Must have at least one digit
- (ii) Must have decimal point
- (iii) Can be either positive or negative
- (iv) Default sign is positive
- (v) No comma's or blanks are allowed with in a constant
- (vi) The range of real constant expressed in exponential form is $3.4 e^{+38}$ to $3.4 e^{-38}$

(a) Valid Real Constants

Some examples of valid real constants are given below.

0.01, 1.5, 825.053,
12E8, 12e8, 12e+8, 12e-8, 0.65E-3

(b) Invalid real constants

Some examples of valid real constants are given below

- | | |
|---------|--|
| 1 | either decimal joint or exponent point must be present |
| 1,000.0 | illegal character (,) |
| 53e10.3 | exponent must be integer cot can't contain a decimal point |
| 13 E 15 | illegal character (blank space) in exponent part |

The quantity 2×10^4 can be represented in any of the following ways

2000 2e4 2E4 2e X 4
0.2e5 .2e5 20e3 20.E+3

Similarly the quantity 8.026×10^{-18} can be represented in any of the following ways

8.026E-18, 0.8026e-17, 80.26c-19 0.00008026e-13

Each floating point constant occupies typically 2 words (8 bytes) memory. Some version of C permits long floating point constant by appending letter (l or L) at the end of constant. (e.g. 0.123456789E-3L). The precision of floating point constants is dependent on compiler used. Virtually all compilers permit at least 6 significant figures and some even permit up to eighteen significant powers.

2.8.7 Character Constants

A character constant is a single alphabet a single digit or a single special symbol enclosed in apostrophes (i.e. Single quotation marks). Here both quotation marks should point to left for example is 'A' valid character constant where as 'A' is not. Most computers and virtually all personal computers make use of ASCII (i.e. American standard code for information interchange) character set, in which each individual character is numerically encoded with its own unique 7-bit combination ($2^7 = 128$ different characters) ASCII table is given in appendix-1. Table in appendix-1 shows the ASCII character set and corresponding decimal equivalent of 7-bit combination. This allows character type data items to be compared with one another. A few examples of character constants and these corresponding values are given in example 2.7

| Constant | Value |
|-------------------|-------|
| 'A' | 65 |
| 'b' | 98 |
| '3' | 51 |
| ' ' (blank space) | 32 |

Example 2.7 Some character constants and their corresponding ASCII values

These values will be same for all the computers that utilize ASCII character set. Their values will be different for the computers that utilize an alternate character set. For example IBM mainframe computers utilize EBCDIC (extended information code) character set, in which each individual character is numerically encoded with its unique 8-bit combination EBCDIC is entirely different from ASCII character set.

2.8.8 String Constants

String constants consist of any number of consecutive characters enclosed in (double) quotation marks. This is shown in example 2.8

"green" "Rs95.50" "Hi \n How are \n you" " "
(empty string)

Example 2.8 String constants

The string "Hi \n How are \n You" is printed as

```
Hi
How are
You ?
```

Because the escape sequence new line character '\n' is embedded in the string. Data types range is given in the below table

| C type | Size (bytes) | Lower bound | Upper bound |
|--------------------|--------------|----------------------------|----------------------------|
| char | 1 | — | — |
| unsigned char | 1 | 0 | 255 |
| short int | 2 | -32768 | +32767 |
| unsigned short int | 2 | 0 | 65536 |
| (long) int | 4 | -2^{31} | $+2^{31} - 1$ |
| float | 4 | $-3.2 \times 10^{\pm 38}$ | $+3.2 \times 10^{\pm 38}$ |
| double | 8 | $-1.7 \times 10^{\pm 308}$ | $+1.7 \times 10^{\pm 308}$ |

Table 2.4 The range of data types

2.9 Variables

A variable is an identifier that may be used to store data value. A value or a quantity which may vary during the program execution can be called as a variable. Each variable has a specific memory location in memory unit, where numerical values or characters can be stored. A variable is represented by a symbolic name. Thus variable name refers to the location of the memory in which a particular data can be stored. Variables names are also called as identifiers since they identify the varying quantities.

For Ex : $\text{sum} = a + b$. In this equation sum, a and b are the identifiers or variable names representing the numbers stored in the memory locations.

Rules to be followed for constructing the Variable names(identifiers)

1. They must begin with a letter and underscore is considered as a letter.
2. It must consist of single letter or sequence of letters, digits or underscore character.
3. Uppercase and lowercase are significant. For ex: Sum, SUM and sum are three distinct variables.

4. Keywords are not allowed in variable names.
5. Special characters except the underscore are not allowed.
6. White space is also not allowed.

2.9.1 Variable Declarations

In the program data types are written as given below.

- | | | | |
|-------|-----------------------|---|--------|
| (i) | Integer | - | int |
| (ii) | Floating point | - | float |
| (iii) | Double floating point | - | double |
| (vi) | Character | - | char |

Assigning an identifier to data type is called type *declaration*. In other words a declaration associates a group of variables with a specific data types. All variables must be declared before they appear in executable statements. The syntax of variable declaration and some examples are given below. The syntax of variable declaration and some examples are given below.

Syntax :

Data type 'variable'

For example:

```
int a; float c; char name;
int sam; double count;
```

If two or more variables of the same data type have to be declared they can be clubbed as shown in example given below.

- i.

```
int a; int b; int c;
this is same as
int a, b, c;
```
- ii.

```
float d; float e; float f;
this can be written as
float d, e, f;
```

```
short int a, b, c ;
long int r, s, t ;
```

The above declarations can also be written as :

```
short a, b, c;
long r, s, t;
```

2.9.2 Assigning Values to Variables

Values can be assigned to variables using assignment operator "=".

Syntax:

Variable name = value;

For Example: :a = 10; int lvalue = 0;

It is also possible to assign a value to a variable at the time the variable is declared. The process of giving initial value to the variable is called initialization of the variable.

For Example:

int a=10, b=5;

float x=10.5, y=1.2e-9

The data item can be accessed in the program simply by referring to the variable name. Data type associated with the variable cannot be changed. However variables hold the most recently assigned data. This is shown in the example2.9 given below.

C program contains the following lines

```
Int a, b, c;
Char d;
- - -
a = 3;
b = 5;
c = a + b ;
- - -
a = 4;
b = 5;
c = a - b ;
```

Example 2.9 nature of variables.

Here a, b, c are integer variables and d is a char-type variable. This type declaration is fixed throughout the program. In the initial lines. Integer quantity 3 is assigned to a, 5 is assigned to b, sum of a and b is assigned to c. character 'a' is assigned to d. these values will be retained till new values are assigned. In the last lines values are redefined to variables as shown in the example 2.9. Integer 4 is assigned to a, replacing earlier value 3; then 2 is assigned to b, replacing earlier value 5; then the difference between a & b i.e. 2 is assigned to c. Replace earlier value 8. finally character 'w' is assigned to d, replacing earlier character d.

2.9.3 Declaring a variable as constant

We may want the value of the certain variable to remain constant during the execution of the program. We can achieve this by declaring the variable with const qualifier at the time of initialization.

For example,

const int tax_rate = 0.30;

The above statement tells the compiler that value of variable must not be modified during the execution of the program. Any attempt change the value will generate a compile time error.

2.10 Questions

1. Give classification of c character set
2. State the rules for identifiers. Are uppercase letters equivalent to lowercase letters? Can digits be included in an identifier name? Can any special characters be included?
3. What are keywords in c ? Can they be used as variable names? Justify your answer.
4. Explain basic data types along with their qualifiers.
5. Give classification of constants and explain them.
6. What are variables and how are they classified ?

2.11 Programing exercise

1. Which of the following are invalid variable names and why?
Determine which of the following are valid identifiers. If invalid, explain why.

- a) record1
- b) \$tax
- c) name-and-address
- d) l r e c o r d
- e) name-and-address
- f) f i l e – 3
- g) name and address
- h) 123-45 -6789
- i) return

2) Write appropriate declarations for each group of variables and arrays.

(a) Integer variables: **p, q**

Floating-point variables: x , y , z

Character variables: a, **by** c

(b) Floating-point variables: root 1 , root2

Long integer variable: counter

Short integer variable: f l a g

(c) Integer variable: index

Unsigned integer variable: cust-no

Double-precision variables: gross, tax, net

(d) Character variables: current, l a s t

Unsigned integer variable: count

Floating-point variable: error

(e) Character variables: `first`, `last`

80-element character array: `message`

3. Find which of the following are valid identifiers. If invalid justify your answer.

- a. `Float`
- b. `1record`
- c. `Do`
- d. `Data of birth`
- e. `Date-of-birth`
- f. `Date_of_birth`
- g. `$num`

4 Determine which of the following numerical values are valid constants. If it is valid specify whether it is real, int or character. If not justify your answer.

- 2.4 `0.72`
- 2.5 `75(d)`
- 2.6 `OX12C`
- 2.7 `OXabc123g`
- 2.8 `O178`
- 2.9 `9.3e12`
- 2.10 `0.9E0.8`
- 2.11 `0.8e+5`
- 2.12 `O127a`

5 Determine which of the following are valid character constants and string constants

- 2.13 `' a '`
- 2.14 `" a "`
- 2.15 `{ a }`
- 2.16 `" tree $"`
- 2.17 `' hello c '`
- 2.18 `' /n '`

6 Write appropriate declaration for each group of variables given and assign the values from the given set.

- 2.19 Integer variable : `num`, `b`, `first`
 - 2.20 Floating point : `d`, `river`, `name`
 - 2.21 Character : `coin`, `reg_no`
 - 2.22 Short integer variable : `tree`
 - 2.23 Long integer variable : `pen`
 - 2.24 Double precision variable : `book`
- { `0`, `8`, `71`, `'e'`, `51.2`, `25e2`, `11.7652`, `0.005`, `2.88X10-12`, `8.5x 105`, `error`, `'b'`}

MODULE 1

UNIT 3. Operators IN C

3.1 Introduction

C has a large number of built in operators. An operator is a symbol which acts on operands to produce certain result as output. The data items on which operators act upon are called '*operands*'. For example in the expression $a+b$; $+$ is an operator, a and b are operands. The operators are fundamental to any mathematical computations.

1. Based on the number of operands the operator acts upon, Operators can be classified as follows:

- a. Unary operators: acts on a single operand. For example: unary minus(-5, -20, etc), address of operator (&a)
- b. Binary operators: acts on two operands. Ex: $+$, $-$, $\%$, $/$, $*$, etc
- c. Ternary operator: acts on three operands. The symbol $?:$ is called ternary operator in C language. Usage: $big = a > b ? a : b$; i.e if $a > b$, then $big = a$ else $big = b$.

Based on the functions, operators can be classified as given below in the table 3.1

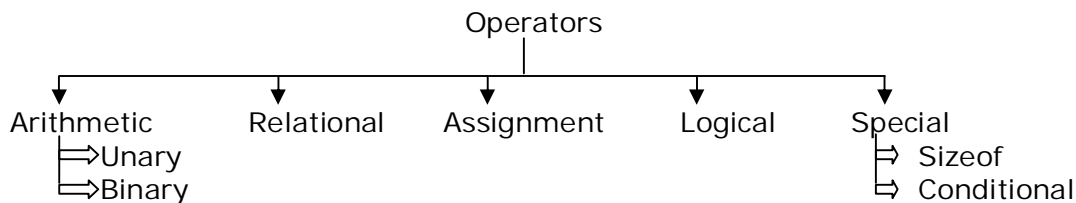


Table 3.1 Classification of Operators

3.2 Arithmetic Operators

Arithmetic operators are used to perform numerical operators in C. They are divided into two classes namely, Unary and Binary arithmetic operators.

Unary operators:

| Operators | Actions |
|-----------|-------------|
| - | Unary minus |
| ++ | Increment |
| -- | Decrement |

Binary operators:

| Operator | Action |
|----------|----------|
| + | Addition |

| | |
|---|----------------|
| - | Subtraction |
| * | Multiplication |
| / | Division |
| % | Modules |

Unary Minus (-)

This symbol is used to indicate the negative sign of the value. For example

```
a = -10;
b = -a;
```

Here the value -10 is assigned to a and the value-a to b.

Increment and decrement operator

The increment operator (+ +) adds 1 to its operand and decrement operator (- -) subtracts one from its operand

To be precise

```
Y = y + 1
    Is written as
Y = y + +
```

These are two types of increment and decrement operators. They are

- Prefix increment (++i)
- Postfix increment (i++)
- Postfix decrement (--i)
- Prefix decrement (--i)

In the prefix increment, first increment and then do operation .In postfix increment, first do the operation and then increment. In prefix decrement first decrement and then do operation. In postfix decrement, first do the operation and then decrement. This is shown in the example 3.1

Let the integer variable ' i ' is variable one.

```
(i) i.e. int i= 1;
    Print f ( " i = %d \n", i);
    Print f ( " i = %d \n", ++i);      //prefix increment
    Print f ( " i = %d /n", i);
```

Output

```
i = 1
i = 2
i = 3
```

Explanation :

In the first statement original value of i is displayed. In the second statement since prefix is used, first i is incremented i.e. $++i = i + 1$, $i = 2$. Then the value is printed hence $i = 2$.

In the third statement, final value is retained and displayed.

```
(ii)  Let i = 1;
      Print f (" i = %d \n ", i );
      Print f (" i = %d \n ", i + +);
      Print f (" i = %d \n " );
```

Output

```
i = 1
i = 1
i = 2
```

Example 3.1 prefixes & postfixes operation.

Explanation: Here in the first statement, the original value of i is 1 is printed. In the second statement since post increment is used. First the operation on of printing is done.

i.e. $i = 1$, then i is incremented. Find I value of $i = 2$, is displayed in the third statement.

(iii) Arithmetic operation

There are five arithmetic operations in C. they are

| Operator | Action |
|----------|----------------|
| + | Addition |
| - | Subtraction |
| * | Multiplication |
| / | Division |
| % | Modules |

Addition, subtraction and multiplication operations are same as in normal decimal operation. The division operator deserves special attention. For division dividend operand must be non zero. Division of one integer by another integer is referred as integer division. This operation results in truncated quotient. (That is decimal portion of the quotient truncated). To avoid this quotient storage operator data type must be float. This is as shown the example 3.2

Case 1 :

```
int a = 5;
int b = 2;
int c ;
```

```
c = a / b;
Print f ( "the quotient c = %d \n", c);
```

Output:
The quotient c = 2

Case 2 :

```
int a = 5;
int b = 2;
float c ;
c = a / b;
Print f ( "the quotient c = %f \n", c);
```

Output :
The quotient c = 2.5

Example 3.2: Integer division

The modulus operator (%) produces the remainder of the division. This is as shown in the example 3.3

```
int a = 20;
int b = 5;
float c, d;
c = a / b;
d = a%b
Print f ( "the quotient = %f \n", c);
Print f ( "the remainder = %f \n", d);
```

Output :
The quotient = 4
The remainder = 0

Example 3.3 Operation of modulus operator

3.3 Relational Operators

| Operator | Action |
|----------|--------------------------|
| < | Less than |
| < = | Less than or equal to |
| > | Greater than |
| = > | Greater than or equal to |

The associativity of these operators is from left to right. The following are the two equality operators associated with the relational operators.

| Operator | Action |
|----------|--------|
|----------|--------|

| | | |
|----------------|----------------|--------------|
| <code>=</code> | <code>=</code> | equal to |
| <code> </code> | <code>=</code> | not equal to |

Relational operators logically compare the values, according to condition given in the program. If the condition is satisfied, this results in '*true*' state otherwise '*false*' state. True is represented by integer value 1 and false is represented by integer value 0. This is shown in example 3.4

Example

Let i, j, k be integer variables, Where the values are 1, 2, 3 respectively.

| Expression | interpretation | Values |
|-----------------------------|----------------|--------|
| <code>i < j</code> | true | 1 |
| <code>i < = j</code> | false | 0 |
| <code>(i + j) > 2</code> | true | 1 |
| <code>j == 2</code> | true | 1 |
| <code>k != 3</code> | false | 0 |

Example 3.4 Relational operators operation.

3.4 Logical Operators

C has three types of logical operators. They are

| Operator | Action |
|--------------------------|--------|
| <code>& &</code> | and |
| <code> </code> | or |
| <code>!</code> | not |

In logical operation the operands are themselves logical. The individual logical expression is compared, using logical operators. The result evaluates which is represented by integer values 0 or 1. The logical AND evaluates to true condition only if both of the operands are true. Logical OR evaluates to true condition if any one of the condition is true. The logical expression can be changed from false to true or from true to false with negation operator. This is as shown in example 3.5

Logical OR (`||`)

| Expression | interpretation | Value |
|-----------------------------|----------------|-------|
| <code>False false</code> | false | 0 |
| <code>False true</code> | true | 1 |
| <code>True false</code> | true | 1 |
| <code>True true</code> | true | 1 |

Logical AND (`& &`)

| Expression | interpretation | Value |
|-----------------|----------------|-------|
| False & & false | false | 0 |
| False & & true | false | 0 |
| True & & false | false | 0 |
| True & & true | true | 1 |

Logical Negation Operator (!)

| Expression | interpretation | Value |
|------------|----------------|-------|
| ! (true) | false | 0 |
| ! (false) | true | 1 |

Example 3.5: AND and OR operators conditions.

The OR and AND operators are used in practical examples as shown in cases of example 3.6

Case 1 :

```
Let a = 4, b = 5, c = 6
    (a < b) && (b < c)
=    (4 < 5) && (5 < 6)
=    true && true
=    true
=    value=1
```

Case 2 :

```
Let a = 4, b = 5, c = 6
    (a < b) || (b > c)
=    (4 < 5) || (5 > 6)
=    true || false
=    true
=    value=1
```

Case 3 :

```
Let a = 4, b = 5, c = 6
    ! (a < b) || (b > c)
=    ! (4 < 5) || (5 > 6)
=    ! (true) || false
=    false
=    value=0
```

Example 3.6: Functioning of OR, AND and NOT operators

3.5 Conditional Operator

C has special operator called ternary or conditional operator. It uses three expressions. The general format of conditional operator is given below.

Expression 1 ? Expression 2 : Expression 3

Evaluation of expression 1 is done logically. Hence it results either in true (i.e. value = 1) or in false (i.e. value = 0) condition. If expression 1 results in true condition then expression 2 is evaluated this becomes the value of expression 1. If expression 1 results in false condition, then expression 3 is evaluated and this value is assigned to expression 1. This is shown in example 3.7

case 1

Let i be integer variable.

int i = 0;

(i < 1) ? 0 : 50

Expression i < 1 is true

Hence the entire conditional expression takes the value 0. Otherwise the value is 50

case 2

Let a, b be integer variables.

int a, b, c;

a = 2, b = 5;

c = (a == b) ? 5 : 10

Expression (a == b) is evaluated and this results in false condition. Hence the conditional expression takes the value 10.

i.e C = 10.

Example 3.7 Function of conditional operator

3.6 The sizeof Operator

The sizeof operator is one of the special operator. This operator returns the size of its operand in bites. The sizeof operator always precedes its operand. This is as shown in the example 3.8

Let int i ; float b; double d; char c; be the variables

Print f("integer requires: %d bytes \n", sizeof i) ;

Print f("float requires: %d bytes \n", sizeof b) ;

Print f("double requires: %d bytes \n", sizeof d) ;

Print f("character requires: %d byte \n", sizeof c) ;

Output

Integer requires : 2 bytes

Float requires : 4 bytes

Double requires : 8 bytes

Character requires : 1 byte

Example 3.8 Operation of sizeof operator.

3.7 Precedence of Operators

The operators have precedence. This is given in table 3.2

| Operator category | Operators | Associativity |
|---|----------------------|---------------|
| unary operators | - ++ ! sizeof (type) | R+L |
| arithmetic multiply, divide and remainder | * / % | L-+R |
| arithmetic add and subtract | + - | L+R |
| relational operators | < <= > >= | L+R |
| equality operators | != | L-+R |
| logical and | && | L-+ R |
| logical or | | L+R |
| conditional operator | ? : | R+L |
| assignment operators | = += -= *= /= %= | R+L |

Table 3.2 Operator Precedence Groups

3.8 Questions

- 1) What is an operand and an operator?
- 2) What is syntax and symantics.
- 3) Give the classification of c language operators.
- 4) Explain with example arithmetic operators?
- 5) Explain unary operators?
- 6) Explain working of the conditional and size of operators
- 7) Illustrate with an example the different between equal to (= =) and assignment operator (=).

3.9 Programing Exercise

- 1) Let a, b, c be integer variables having values 2, 4, 5 respectively and x, y, z be float variables having values 1.1, 2.5, 3.6 respectively. Determine the value of each arithmetic statements.
 1. $a + b + c$
 2. $a \% b$
 3. a / b
 4. $a * (b / c)$
 5. $x + y + z$
 6. $x / (y + z)$
 7. $x \% y$

4 Determine the value of each statement.

Let $x = 2$, $y = 5$, $z = 10$

2.4 $p = (x == 2)? y : z$

2.5 $q = (x >= 0)? y : z$

MODULE 2

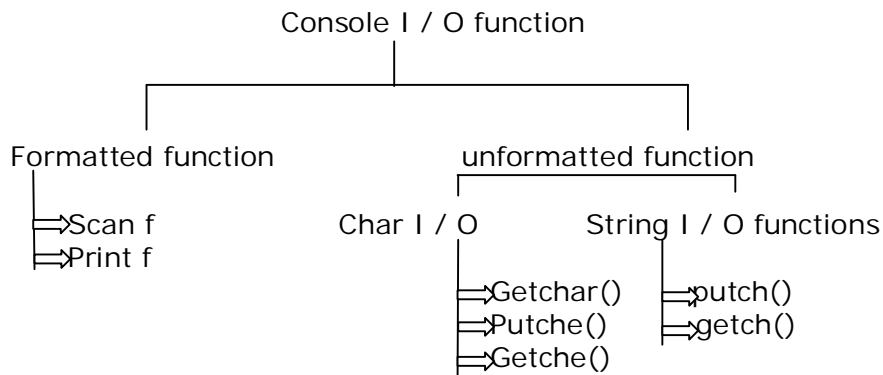
Unit 1 : Input Output Statements

1.1 Introduction

In c language input and output functions are accomplished through library function. The header file for I/O function is `<stdio.h>`. In C there are two types of I/O functions. They are console I/O and file I/O. In this chapter only console I/O functions are dealt. Console I/ o function takes I/P from keyboard and produces o/p on the screen. The console Input / Output functions are also called as *Standard input / output functions*.

Classification:

The console I / O function are classified as shown below.



1.2 Formatted I / O Functions

Formatted I / O means reading and writing data in formats which are desired by the user. The function used to input action is `scanf()` and output action is `printf()`.

1.3 The `printf()` Function

`Printf()` is used to print or display data on the console in a formatted form. The format of `printf()` is `printf("control string", list of arguments);` Control string contains the formatting instructions within quotes. This can contain

- (i) characters that are simply printed as they are
- (ii) Conversion specifications with begins with format specifier (%) sign.
- (iii) Escape sequences.

Arguments values get substituted in the place of appropriate format specifications. Arguments can be variables, constants, arrays or complex expressions. The percentage (%) followed by conversion character is called format specifier. This indicates the type of the corresponding data item.

In the table 1.1 most frequently used format specifiers are listed.

| Format specifier | Meaning |
|------------------|--|
| % d or % i | decimal integers |
| % u | unsigned decimal integer |
| % x | unsigned hexadecimal (lower case letter) |
| % X | unsigned hexadecimal (upper case letter) |
| % o | octal |
| % c | character |
| % f | floating point |
| % s | strings |
| % lf | double |
| % ld | long signed integer |
| % lu | long unsigned integer |
| % p | displays pointer |
| % % | prints a % sign |
| % e | scientific notation (e lower case) |
| % E | scientific notation (e upper case) |

Table 1.1 Format specifiers

1.3.1 Application of `printf()` Function

The `printf()` function is used to print the different types of output. This is given below.

- (i) Printing given data
- (ii) The printf() statement can be used without format specifier, just to print the given data. This is as shown below.

```
printf( " C programming is easy" );
```

Output:

C programming is easy

- (iii) Printing numbers

To print integers %d or %i is used to print floating point %f is used. This is as shown below.

```
int a = 5
```

```
float d = 10.3;
```

```
printf( "the value of integer a is %d", a );
```

```
printf( "the value of float d is %f", d );
```

Output:

The value of integer a is 5

The value of float is 10.3

In the above instance instead of using two printf() statements only one printf() can be used. This is as shown below

```
printf( "the value of integer a is %d \n, the value of float d is %f" a,d);
```

Output

The value of integer a is 5

The value of float is 10.3

- (iv) Printing character / string data

To print character %c used and to print a string %s format specifier is used. This is shown below.

```
printf( "this is %c %s", 'a', "book");
```

The characters are always given in single quote and strings are enclosed in double quote.

1.3.2 Escape Sequence in printf() Function

In addition to format specification, escape sequence can also be used in printf(). These are specified in the control string portion of the printf() and are used mainly for screen formatting. All escape sequences are

provided with slash (/). Since back slash is considered as an “*escape*” character. These sequences are called escape sequences. These sequences cause an escape from the normal interpretation of a string. So that the next character (after blank slash) is recognized as having a special meaning.

- Usage of \t moves the cursor to next tab problem
- \n makes the a cursor to go to new line.
- \r moves the cursor to the beginning of the line in which it is currently placed.
- \a alters the user by sounding the inbuilt speaker of system.
- \b moves the cursor one position to the left of its current opposition.
- The character of single quote and back slash can be printed by using escape sequence \', \", \\ respectively. This is as shown in example 3.1

(i) `printf("Teacher asked, \" did you understand ?\" ");`
 This will print
 Teacher asked, "did you understand?"

(ii) `printf("The sequence is a<b, where \'b\' is > 0");`
 This will print
 The sequence is a<b, where 'b' is >0

(iii) `printf("Hi,\n How are \n You?");`
 This will print

Hi
 How are
 You

Example 1.1 Escape Sequence Usage

1.3.3 Minimum Field Width Specifier (MFWS)

An integer placed between the % sign and format code is called minimum field width specifier or % number format code; ex. %5d. If string or number to be printed is longer than that minimum it will printed as such. Minimum field width specifiers are used to make the O/P such that it reaches the minimum length specified. This is done by padding. Default padding is done with spaces. This is used most commonly produce table in which column line up.

If we wish to pad the space (if any) with zeros, place zero before MFWS. For example %d, will pad a number with 'zeros' if numbers is less than five digits. So that the total length is 5. %d will pad a number with 'apace'. If number is less than five digits. So that the total length is five.

The following example demonstrates the working of minimum field width specifier.

```
Double count;
Count = 10.51324;    // here total digits are 8, includes dot ( . )
operators.
```

```
printf (" %f" , count);
printf (" %10f" , count);
printf (" %010f" , count);
```

Output

```
10.51324
10.51324    // inserts 2 blank spaces. So that total length = 10
digits
0010.51324    // inserts 2 zeros. So that total length = 10 digits
```

Example 1.2 Minimum field width specifier

1.3.4 The Precision Specifier

Precision specifier follows minimum field width specifier (if given). It consists of a period followed by an integer (i.e. for ex %5.1d). the way it works depends upon data types used. When precision is applied to float data, it determines the number of decimal places displayed. For example % 8.5f displays a number at least 8 character width with four decimal places. When precision is applied to string, the precision specifier species maximum field length. For example %5.9S, displays a string at least five and not exceeding of characters long. If string is longer than the maximum field width the end character are truncated, so that string length is equal to integer types. The precision specifier determines the minimum number of digits. That appears for the given number. Leading zeros are added to achieve the required number of digits. This is shown in the example 1.3.

```
printf("5.4f\n", 176.012543871);
printf("2.8d\n", 5214);
printf("8.12S\n", "This is C language book");
```

Output

```
176.012543871
00005214
This is C language book
Example 1.3: Precision specifier
```

1.4 The scanf() Function

The `scanf()` reads the input data from standard input device. i.e. keyboard. The general format of the `scanf()` function is `scanf("format string" list of arguments);` where format string consists of format specifiers and arguments consists of address of variables. To this corresponding address the data read from keyboard is sent. The address of the variable is denoted by ampersand symbol '&' (it is called as "address of the operator). Some format specifiers as shown in the table 1.1 is used. Example 1.4 shows how to work with `scanf()` function.

Note that the values that are supplied through keyboard must be separated by either blank tabs or newlines. Escape sequences are not included in `scanf()` function.

(i) To read integer data:

```
int i ;
- - -
scanf("%d", &i);
```

(ii) To read floating point data:

```
floatf;
- - -
scanf("%f", &f);
```

(iii) To read character data;

```
char sam, john;
- - -
scanf("%c", &sam, &john);
```

(vii) To read more than one data types at a time

```
int i;
float b;
char c;
string s;
- - -
scanf("%d %f %c %s", &i, &b, &c, &s );
```

Example 1.4: `scanf()` function

1.5 Unformatted I / O Function

A simple reading of data from keyboard and writing to I / O device, without any format is called unformatted I / O functions. This is classified as string I / O and character I / O.

1.6 Character Input/Output(I/O)

In order to read and output a single character character I / O functions are used. The functions under character I / O are

- (I) getchar()
- (II) putchar()
- (III) getch()
- (IV) getche()

The getchar() function

Single characters can be entered into the computer using the C library function getchar. The getchar function is a part of the standard C I/O library. It returns a single character from a standard input device typically a keyboard. The function does not require any arguments, though a pair of empty parentheses must follow the word getchar.

The general syntax

Character variable=getchar();

Where character variable refers to some previously declared character variable. Example:

A C program contains the following statements

```
char c;
```

```
.....
```

```
C=getchar();
```

The first statement declares that c is a character-type variable. The second statement causes a single character to be entered from the standard input device and then assigned to c. If an end-of-file condition is encountered when reading a character with getchar function, the value of the symbolic constant EOF will automatically be returned. (This value will be assigned within the stdio.h file. Typically, EOF will be assigned the value -1, though this may vary from one compiler to another).

The getchar function can also be used to read multi character strings, by reading one character at a time within a multi pass loop.

```
int i, j;
char a, b;
i = 65;
b = 'd'
j = getchar();           // I / O int data type into j
a = getchar();           // I / O char data type into a
putchar(i);
putchar(b);
putchar(a);
putchar(j);
```

Output

```
a
d
```

--- (according to user I / O)

Drawback with `getchar()` is that buffers the input until 'ENTER' key is pressed. This means that **getchar** does not see the characters until the user presses return. This is called line buffered input. This line buffering many leave one or more characters waiting in the input queue, which leads to 'errors' in interactive environment. Hence alternatives to `getchar()` is used.

The `putchar()` function:

Single characters can be displayed using the C library function `putchar`. This function is complementary to the character input function `getchar`. The `putchar` function, like `getchar`, is a part of the standard C I/O library. It transmits a single character to a standard output device. The character being transmitted will normally be represented as a character type variable. It must be expressed as an argument to the function, enclosed in parentheses, following the word `putchar`.

The general syntax is

```
putchar(character variable)
```

where character variable refers to some previously declared character variable.

A C program contains the following statements

```
Char c;
```

```
.....
```

```
putchar(c);
```

If `putchar()` is called with integer value, the equivalent ASCII character is displayed.

Alternatives to `getchar()`

The two common alternative functions to `getchar()` are

```
getch()
```

```
getche()
```

The `getch()` function reads a single character at a time and it waits for key press. It does not echo the character on the screen. The `getche()` is same as `getch()`/ but the key is echoed. This is illustrated below.

```
char ch;
```

```
Ch = getch()
```

```
// let key press = k
```

```
putchar(ch);
```

Output

1.7 String I / O

In order to read and write string of characters the functions `gets()` and `puts()` are used `gets()` function reads the string and `puts()` function takes the string as argument and writes on the screen. This is illustrated below

```
char name [50]
puts ("Enter your name");
gets (name);
puts(" The name entered is ")
```

Output:

```
Enter your name: Fredric      /* string as entered by user*/
The name entered is: Fredric
```

Note: Char name (50) is character array declaration. This is explained in unit - Arrays

1.8 Questions

1. What are the commonly used input I output functions in C?
2. Explain the instructions to read and write a single character
3. What is the purpose of the `printf()` function? How is it used within a C program? Compare with the `putchar()`
4. What is the purpose of the `scanf()` function? How is it used within a C program? Compare with the `getchar()`
5. If escape sequences are given in the control string of `printf()` statement how are they interpreted in the output?
6. Explain the difference with example `getche()`, `getchar()`, `getch()`.

1.9 Programming Exercises

1. A C program contains the following statements:


```
#include <stdio.h>
char a, b, c;
```

 - (a) Write appropriate `getchar` statements that will allow values for a, b and c to be entered into the computer.
 - (b) Write appropriate `putchar` statements to output the values of a, b and c
2. A C program contains the following statements:


```
#include <stdio.h>
Variables - i, j , k;
```

Write an appropriate `scanf` function to enter numerical values for i, j and k, assuming

- (a) The values for i, j and k will be integers.

- (b) The value for i = decimal integer, j =floating point and k = character.
 (c) The values for l and j = long integer and k =double floating point

3. Write a programme to find the area of a square

A C program contains the following statements:

```
#include <stdio.h>
int i, j;
long sum;
short scale;
unsigned U;
float data;
double data1;
char name;
```

For each of the following groups of variables, write a scanf function that will allow a set of data items to be read into the computer and assigned to the variables. Assume that all integers will be read in as decimal quantities.

- (a) i,j, sum and scale (c) i,data and name
 (b) i, j and U (d) data1,name,data1

4. A C program contains the following statements.

```
#include <stdio.h>
char x, y, z;
```

Suppose that \$ is to be entered into the computer and assigned to a, * assigned to b and @ assigned to c. Show

how the input data must be entered for each of the following scanf functions.

- (a) scanf ("%c%c%c", &x, &y, &z) ;
 (b) scanf("%c %c %c", &x, &y, &z);

5. A C program contains the following statements:

```
#include <stdio.h>
int a, b;
long ia;
unsigned d;
float p;
double q;
char c;
```

For each of the following groups of variables, write a printf function that will allow the values of the variables to be displayed. Assume that all integers will be shown as decimal quantities.

- (a) a, b, and d (c) a,ia and c
 (b) a, q,ia,

A C program contains the following statements:

```
#include <stdio.h>
int i, j;
long ix;
unsigned U;
float x;
double dx;
```

```
char c;
```

Write an appropriate printf function for each of the following situations, assuming that all integers will be displayed as decimal quantities.

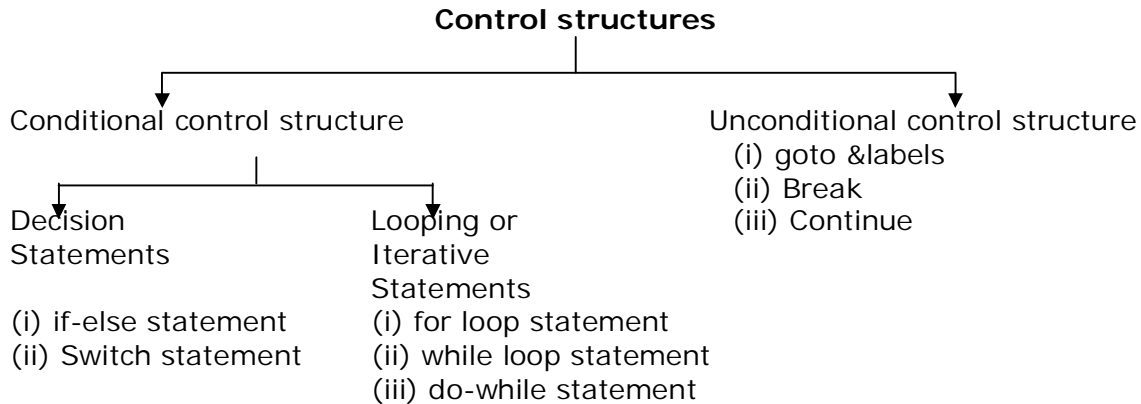
```
* * * * *
```

MODULE 2

UNIT 2: Control Structures

2.1 Introduction

In most of C programs that are encountered so far, the instructions appeared in serial fashions. In reality C program requires a logical test to be carried at some point in the program depending upon the outcome of the logical test, one of several possible actions can be carried out. This is called *branching*. The branching structure which controls the program flow is called control structure. The classification of control structures is given below.



An expression such as `x=0`, `printf()`, or `a=b+c` becomes 'statements'. When it is terminated by semicolon, as shown below.

```

x=0;
printf(. . .);
a=b+c;
    
```

Braces { and } are used to group statement into a block. So that they are syntactically equivalent to a single statement. These braces are used to group multiple statements in if, else, structures, while or for loops. In the conditional controlled structure, a condition is given. This condition is logically evaluated. Depending upon whether the condition evaluates to either true or false state. The appropriate branching is taken. Unconditional branching means transferring the control of the program to a specified statement without any conditions. Let us study each one in detail.

2.2 The if – else statement

The if – else statement contains a condition. It is logically evaluated. If it evaluates to TRUE condition, then statements under if part is executed. If the condition evaluates to FALSE state then statements under else part is executed. *Else* clause is optional. The statement under if or else part can be single or multiple statements. If multiple statements are used, they

must be embraced with the braces. The syntax of if – else statement is given here.

Syntax: i) if (expression)
 Statement;
 else
 statement; /* else part is optional */

 ii) if (expression)
 {
 Statement1;
 Statement2;
 . . .
 Statement n;
 }
 else
 {
 Statement1;
 Statement2;
 . . .
 Statement n;
 }

Some valid forms of if are given below.

- (i) if (x > 0)
 printf("%i", x);
- (ii) if (a > b)
 printf(" a is greatest ");
 else
 printf(" b is greatest ");
- (iii) if (a == b)
 {
 printf(" a & b are equal ")
 printf(" x/a is equal to x/b ")
 }

Some invalid forms of if are given below.

- (i) if (a > b);
 printf("a is greatest")

the semicolon at the end of condition, makes if (a >b) as invalid.

- (ii) if (a > b)

```
statement1;
statement2;
statement3;
```

Here multiple statements under if are not enclosed in braces.

```
(iii) if ( a > b )
    {
        Statement 1;
        statement 2;
        . . .
        Statement n;
    };
```

Here the ending brace has semicolon (;)

Usage of if else condition is illustrated in the following examples1 and 2.

Example1

Write a program to calculate greatest of 3 numbers.

```
/* Write a program to calculate greatest of 3 numbers */
```

```
# include <stdio.h>
main()
{
    int a, b, c;
    printf("Enter three numbers");
    scanf (" %d %d %d ", &a, &b, &c);
    if ( ( a > b ) && ( a > c ))
        printf ("\n a is greatest");
    if ( ( b > a ) && ( b > c ))
        printf ("\n b is greatest");
    if ( ( c > a ) && ( c > b ))
        printf ("\n c is greatest");
}
```

Example2

Write a program to check a given number is odd or even.

```
/* program to check given number is either odd or even*/
```

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

```

        printf("Enter the number to bechecked");
        scanf (" %d" , &a);
        if ( (a%2) == 0)
            printf("The given number is even");
        else
            printf("The given number is odd");
    }

```

2.3 Nested -ifs

Inside one if statement, if or if-else can be used. Such statements are called nested if's the structure is given below.

```

if( a )
{
    if ( b )
    {
        Statement1;
        - - -
        Statement n;
    }
    if ( c )
    {
        Statement1;
        - - -
        Statement n;
    }
    else
    {
        Statement1;
        - - -
        Statement n;
    }
}
else
{
    Statement1;
    - - -
    Statement n;
}

```

The final else is associated with if (a) and not associated with if c, because it is not in the same block.

2.3.1 The if -else - if Ladder

A common programming construction is the if-else-if staircase because of its appearance. The general form is

```

if ( expression )
    Statement:
else
    if ( expression )
        Statement:
    else
        if ( expression )
            Statement:
        . . .
        else
            Statement:

```

The condition are evaluated from top, towards down. As soon as a true condition is found, the statement associated with it is executed and the rest of the ladder is bypassed. If none of the conditions are true, the final else is executed. If final else is not present, no action taken place if all other conditions are false. This is given in the below example

```

#include <stdio.h>
main()
{
    int i;
    i = 0;
    clrscr();
    printf("Enter your choice (1-4);");
s   scanf (" %d" , &i);
    if ( i == 1)
        printf("\n your choice is 1");
    else if ( i == 2)
        printf("\n your choice is 2");
    else if ( i == 3)
        printf("\n your choice is 3");
    else if ( i == 4)
        printf("\n your choice is 4");
    else
        printf("\n invalid choice");
}

```

Output:

```

Enter your choice (1-4): 2
Your choice is 2.

```

Explanation: here the second condition (i ==2) is true. Hence the following printf() is executed and control will come out of loop. If none of the conditions is satisfied then final the statement is executed.

2.4 Switch Statement

C has a multiple branch selection statements called switch. It tests the value of an expression against a list of integer or character constants. When a match is found the statements associated with that constants are executed. The general format of switch is

```
switch (expression)
{
    case constant1;
        Statement sequence
        break;
    case constant2;
        Statement sequence
        break;
    case constant3;
        Statement sequence
        break;
    . . .
    default
        Statement sequence
}
```

Switch, case and default are key words and statement sequence can be single statement or a compound statement. If it is compound statement it must be enclosed in braces.

The expression which is in switch, must evaluate to integer type. The expression is switch can have int constant or char constant or expression that evaluates to one of their constants. If character data type is used, its equivalent integer value is considered. Floating point expressions are not allowed in switch statement.

In case expression such as case constant1, case constant2, case constant3, . etc the word constant 1 is called case labels or case prefixes. The case label can either be integer or character constant. Each case label must be distinct.

When a switch statement is executed, the expression is evaluated and control is directly transferred to the group of statements where case-label value matches with the value of expression statements under that case label is executed the break statement ensures immediate exit from the switch statement. If none of the cases are satisfied, the default statement is executed. The default statement is optional. If none of the case are satisfied and default case is also not present, then no action takes place. To understand the functioning of switch-case statement a few programs are given below.

Example 1 write a program to check whether the enter the character is a vowel, a last alphabet or consonant

/* program to check whether the enter the character is a vowel, a last alphabet or consonant */

```
#include <stdio.h>
main()
{
    char ch();
    printf("\n Enter a lower case alphabeet (a-z);");
    scanf (" %d" , & ch);
    if ( ch<'a' || ch>'z')
        printf("\n Character is not lower case alphabet");
    else
        switch (ch)
        {
            case 'a' :
            case 'e' :
            case 'i' :
            case 'o' :
            Case 'u' :
                printf("\n Entered Character is vowel");
                break;
            case 'z' :
                printf("\n Entered Character is last alphabet");
                break;
            default;
                printf("\n Entered Character is consonant");
        }
}
```

Output: Enter a lower case alphabet (a – z): p entered character is consonant.

Explanation: each case need not have its own statements. A set of case can have common statements to be executed. This is as shown in the above program. The default can either end with break or not. This depends upon program structure.

```
#include <stdio.h>
main()
{
    int a();
    printf("\n Enter a an integer (10-13);");
    scanf (" %d" , & a);
    switch (a)
```

```

{
    case 10 :
        printf("\n You have entered 10");
        break;
    case 11 :
        printf("\n You have entered 11");
        break;
    case 12 :
        printf("\n You have entered 12");
        break;
    case 13 :
        printf("\n You have entered 13");
        break;
    default:
        printf("\n Invalid choice");
}
}

```

Output :

```

Enter an integer : 11
You have entered : 11

```

Note : observe here the case labels. This in accordance with the program flow labels need not be always in the order of 1, 2, 3 . . etc.,

1. write a program to choose color using switch case statements.

/* a program to choose color using switch case statements */

```

#include<stdio.h>
main()
{
    int a;
    printf(" choose the color u like; red-11, blue-22, violet-13: ");
    scanf("%d ", &a);

    switch(a)
    {
        case 11:
            printf("/n you have selected RED colour ");

        case 22:
            printf("/n you have selected BLUE colour ");

        case13:
            printf("/n you have selected VIOLET colour ");

        default:
            printf("/n Invalid choice");
    }
}

```

```
    }
}
```

Output:

```
choose the color u like; red-11, blue-22, violet-13:  22
                                     you have selected BLUE colour
                                     you have selected VIOLET colour
                                     Invalid choice
```

Analysis:

Observe here the out put. Since the user has entered the choice as 22 , expected out put is only the line *you have selected BLUE color*. But the program has executed the case statements where a match is found and all subsequent cases and default also. The reason is *break* clause is not used. If break clause is used after each case , the problem is eliminated. This is shown in the below example.

/* a program to choose color using switch case statements */

```
# include<stdio.h>
main()
{
    int a;
    printf(" choose the color u like; red-11, blue-22, violet-13: ");
    scanf("%d ", &a);

    switch(a)
    {
        case 11:
            printf("/n you have selected RED colour ");
            break;

        case 22:
            printf("/n you have selected BLUE colour ");
            break;

        case13:
            printf("/n you have selected VIOLET colour ");
            break;
        default:
            printf("/n Invalid choice");
    }
}
```

Output:

```
choose the color u like; red-11, blue-22, violet-13:  22
you have selected BLUE colour
```

2.4.1 Rules of Switch Statement

- The switch variable must be an integer or character type.
- Case labels must be constants of constant expressions.
- Case labels must be unique. No two labels can have the same value.
- Case labels must end with a colon.
- The break statement transfers the program control out of the switch block.
- The break statement is optional. When the break is not written in any 'case' then the statements following the next case are also executed until the 'break' is not found.
- The default case is optional. If present, it will be executed when the match with any 'case' is not found.
- There can be at most one default label.
- The default may be placed anywhere but generally written at the end.
- When placed at end it is not compulsory to write the 'break' for it.
- We can nest the switch statements.

2.4.2 Difference Between if Statement and Switch Statement

| No. | If statement | Switch statement |
|-----|---|--|
| 1 | It checks whether the condition is true or false | It is multi-way decision statement |
| 2 | It contains the condition which will evaluates to true or false | It contains the variable whose value is to be checked |
| 3 | It supports all types of conditions | It supports checking only integer and character values |
| 4 | Syntax: if (condition) { //statements } else { //statements } | Syntax: switch (variable) { case value1: break ; case value2: break ; default : |

2.5 Questions

- 1) Describe if-else statement.
- 2) In what way does switch statement differ from an if statement
- 3) Explain nested if-else statement execution
- 4) Write syntax for nested if-else statement.

2.6 Programming Exercises

1. Write a program to input two numbers from keyboard and find the result of subtraction of greater number – smaller number.
2. Input a year from keyboard and determine whether it is leap year or not.
- 3) Create a menu driven program to input a character from user and Determine:
 - i. Whether it is alphabet or not.
 - ii. Whether it is in upper – case or not. Using switch case.
- 4) Write a program to compute net amount from the given quantity purchased and rate per quantity. Discount @10% is allowed if quantity purchased exceeds 100. Use the formula
Net Amount = (Quantity Purchased x Rate Per Quantity) – Discount
- 5) Write a program to compute the division from the given marks of 5 subjects. The division can be calculated as follows:

| | |
|---|----------|
| 7) Write a program to compute the division from the given marks of 5 subjects. The division can be calculated as follows: - | Division |
| Average Mark | |
| >=60 | First |
| >=50 | Second |
| >=40 | Third |
| <40 | Fail |

MODULE 2

Unit 3: PROGRAM LOOPING

3.1 Loops

In the programming often there is a need to perform an action over and over ie repeatedly, often with variations in the details each time we repeat. This repetitive operation is done through the control structure called **loop** or **iteration**. There are three types of looping, namely:

1. for loop
2. while loop
3. do-while loop

3.2 The for loop

A *for loop* is a block of code that iterates a list of commands as long as the loop control condition is true. The syntax of for loop is

```
for(initialization; Test condition; loop expression)
    statement sequence
```

Syntax explanation:

(1) Initialization:

This is some kind of expression which initializes the control variable or (index variable) This statement is carried out only once before the start of the loop. e.g. `i = 0;`

(2) Condition:

The condition is evaluated at the beginning of every loop and the loop is only carried out while this expression is true. e.g. `i < 20;`

(3) Loop expression

This is some kind of expression for altering the value of the control variable. This expression is evaluated at the end of each iteration. In C it can be absolutely anything. e.g. `i++` or `i *= 20` or `i /= 2.3 ...`

(4) Statement sequence

Statement sequence can consist only one statement or a group of statements. If it contains a group of statements, they must be embraced in braces(`{}`). If it contains only one statement, braces need not be enclosed.

Simple programs using for loop is given below

1. Write a program to print a word *hello* five times using for loop.

/ A program to print hello five times */*

```
#include<stdio.h>
int main()
{
    int i;
    for(i=0; i<5; i++)

        printf("Hello \n");

}
```

Output: Hello
 Hello
 Hello
 Hello
 Hello

Analysis:

Observe the program carefully. Inside the body of the `main()` function, the statement declares an integer variable `i`. In the for statement, the first expression is initialization statement i.e. `i=0`, expression that initializes the integer variable `i` to 0.

The second expression which is conditional expression in the for statement is `i<5`. This expression returns TRUE value(=1) for `i` as long as the relation indicated by the less-than operator (`<`) holds. As mentioned earlier, the second expression is evaluated by the for statement each time after a successful looping. If the value of `i` is less than 5, which means the relational expression remains true, the for statement will start another loop. Otherwise, it will stop looping and exit.

The third expression in the for statement is `i++` in this case. This expression is evaluated and the integer variable `i` is increased by 1 each

time after the statement inside the body of the for statement is executed. In other words, when the for loop is first encountered, i is set to 0, the expression $i < 5$ is evaluated and found to be true, and therefore the statements within the body of the for loop are executed. Following execution of the for loop, the third expression $i++$ is executed incrementing i to 1, and $i < 5$ is again evaluated and found to be true, thus the body of the loop is executed again. The looping lasts until the conditional expression $i < 5$ is no longer true. Summary of evacuation of for statement is as given in the table below.

| Loop step | Initialization expression $i=0$ | Test condition $i < 5$ True/False | Loop expression $i++$ | Output |
|-----------|---------------------------------|-----------------------------------|-----------------------|--------|
| 1 | $i=0$ | True | $i=1$ | Hello |
| 2 | $i=1$ | True | $i=2$ | Hello |
| 3 | $i=2$ | True | $i=3$ | Hello |
| 4 | $i=3$ | True | $i=4$ | Hello |
| 5 | $i=4$ | True | $i=5$ | Hello |
| 6 | $i=5$ | False | Stops execution | |

2. Write a program to convert 0 through 15 integers to hex numbers.
 /* program to convert 0 through 15 integers to hex numbers */

```
#include <stdio.h>
main()
{
    int i;
    printf("Hex(uppercase)\t Hex(lowercase)\t Decimal\n");
    for (i=0; i<16; i++)
        printf("%X %x %d\n", i, i, i);
}
```

OUTPUT

| | Hex(uppercase) | Hex(lowercase) | Decimal |
|---|----------------|----------------|---------|
| 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 |
| 2 | 2 | 2 | 2 |
| 3 | 3 | 3 | 3 |
| 4 | 4 | 4 | 4 |
| 5 | 5 | 5 | 5 |
| 6 | 6 | 6 | 6 |
| 7 | 7 | 7 | 7 |
| 8 | 8 | 8 | 8 |
| 9 | 9 | 9 | 9 |
| A | a | | 10 |
| B | b | | 11 |
| C | c | | 12 |
| D | d | | 13 |
| E | e | | 14 |
| F | f | | 15 |

ANALYSIS

Observe the program carefully. Inside the body of the main() function, the statement declares an integer variable i. The printf statement declares format of the output on the screen. In the for statement .the first expression is initialization statement ie i=0, expression that initializes the integer variable i to 0.

The second expression which is conditional expression in the for statement is i<16. This expression returns TRUE value(=1)for i as long as the relation indicated by the less-than operator (<) holds. As mentioned earlier, the second expression is evaluated by the for statement each time after a successful looping. If the value of i is less than 16, which means the relational expression remains true, the for statement will start another loop. Otherwise, it will stop looping and exit.

The third expression in the for statement is i++ in this case. This expression is evaluated and the integer variable i is increased by 1 each time after the statement inside the body of the for statement is executed. In other words, when the for loop is first encountered, i is set to 0, the expression i<16is evaluated and found to be true, and therefore the statements within the body of the for loop are executed. Following execution of the for loop, the third expression i++ is executed incrementing i to 1, and i<16 is again evaluated and found to be true, thus the body of the loop is executed again. The looping lasts until the conditional expression i<16 is no longer true.

. The statement contains the printf() function, which is used to display the hex numbers (both uppercase and lowercase) converted from the decimal values by using the format specifiers, %X and %x.

3. Write a program to find simple interest for three sets of input.

```
/* calculation of simple interest for 3 sets */
```

```
main( )
{
int p,n, count;
float r, si;
for(count = 1; count <=3; count = count +1)
{
printf("Enter values of p,n and r");
scanf("%d %d %f", & p, & n, & r);
si = p * n * r / 100;
printf("Simple interest = Rs %f \n", si);
}
```

```
    }
}
```

4. Write a program to print upper case and lower case alphabets using using char data type in for loop.

/* a program to print upper case and lower case alphabets using using char data type in for loop */

```
#include <stdio.h> /* library header */
main()
{
    char ch;

    for (ch = 'A' ; ch <= 'z' ; ch++)
        printf("%c\n" , ch);
}
```

This program demonstrates the for loop.

3.2.1 The for Loop Variations

For loop is a powerful programming tool. Many variations can be done in the for loop. They are listed below.

1) for(i=1; i<4; i=i+1)

Here in the initialization part instead of i=i+1, the statements i++ or i+=1 can also be used.

2) for(i=1; i<4;)

```
{
    Printf(" helo for loop");
    i=i+1;
}
```

Here in the for loop statement increment is not given, but given in the body of the statement. Semicolon after the condition is mandatory.

3) int i=2;

for(; i<4; i=i+1)

```
    Printf(" helo for loop");
```

Here in the for loop statement initialization is not given, but given before the statement. Semicolon before the condition is mandatory.

4) int i=2;

for(; i<4;)

```
{
```

```

    Printf(" helo for loop");
    i=i+1;
}

```

Here in the for loop statement initialization and increment is not given. But both the semicolons are mandatory.

3.2.2 The Infinite Loop

In the for loop if conditional expression is absent, the condition is assumed to be true. This turns out to be infinite loop. This is as shown below.

```
for( ; ; )
```

If break statement is used inside the loop , infinite loop is terminated. More about break is discussed in section 3.5 of this unit.

3.2.3 The Comma Operator in The for Loop

The comma (,) operator is basically used in conjunction with *for* loop statement. This permits two expressions to be used in initialization and count section of the for loop. Only one test expression is allowed in the for loop. The syntax of this is given below.

```
for (expression1a, expression1b; expression2; expression3a, expression3b)
```

Here expression 1a and expression 1b and expression 3a expression 3b are separated comma operator.

For example consider the below for statement

```
for(i=0,j=2;j<=0;j--,i++)
```

Here i and j are the two different variables used and they are separated by comma operator. Operation of this is given in the below example:

```

#include <stdio.h>
main()
{
    int i, j;

    for (i=0, j=1; i<8; i++, j++)
        printf("%d - %d = %d\n", j, i, j - i);
}

```

OUTPUT:

```

1 - 0 = 1
2 - 1 = 1
3 - 2 = 1
4 - 3 = 1
5 - 4 = 1
6 - 5 = 1
7 - 6 = 1

```


$$8 - 7 = 1$$

3.2.4 Declaring Variables Inside The for Loop

In the initialization section of the for loop, Variable can be initialized also. The scope of it limited to the block of code controlled by that statement. Consider the example given below.

```
#include<stdio.h>
main()
{
    int j;
    for(int i=0; i<5;i++)
    {
        j=i*i;
        printf("The value of j is%d",j);

    }

    i=i+9;    /*ERROR. i is not known here */
}
```

In the above example the integer variable i is declared inside the for loop in the initialization part of it.. This i is limited only to that declared loop. Outside the loop is unknown. If declared compiler shows error message.

3.2.5 The Nested for Loop

The concept of using a loop within a loop is called *nested loop*. If a for loop contains another for loop statement, such loop is called *nested for loop*.. A nested for loop can contain any number of for loop statements within itself. Usually only two loops are used. In this the first loop is called outer loop and the second is called inner loop. These types of loops are used to create matrix. In this the outer loop is used for counting rows and the internal loop is used for counting columns. The syntax and example program is given below.

SYNTAX:-

```
for (initializing ; test condition ; increment / decrement)
{
    statement;
    for (initializing ; test condition ; increment / decrement)
    {
        body of inner loop;
    }
    statement;
}
```

Example: Write a program to generate a matrix of order 4*4 containing symbol*(asterisk).

```

/*program to generate a matrix of order 4*4 containing
symbol*(asterisk)*/
#include
void main ( )
{
    int i, j;
    clrscr ( );
    for (j=1; j<=4; j++) /*outer for loop*/
    {
        for (i=1; i<=5; i++) /*inner for loop*/
        {
            printf ("*")
        }
        printf ("\n");
    }
    getch ( );
}

```

OUTPUT OF THIS PROGRAM IS

```

*           *           *           *
*           *           *           *
*           *           *           *
* * * *

```

3.3 While Loop

The simplest of the three loops is the *while loop*. The while-loop has a condition and the statements. The syntax is given below.

```

while (condition)
{
    statements;
}

```

The condition expression (something like (a > b) etc...) is evaluated first. If the expression evaluates to True state i.e returns a nonzero value (normally 1) the looping continues; that is, the statements inside the statement block are executed. After the execution, the expression is evaluated again. The statements are then executed one more time if the expression still returns nonzero value. The process is repeated over and over until the expression returns 0(False state).

The statement block, surround by the braces { and }. If there is only one statement in the statement block, the braces can be discarded. Consider an example of using the while statement.

- 1) Write a program to understand the use of while loop.

```
/* program Using a while loop*/
```

```
#include <stdio.h>
main()
{
    int a;
    printf("Enter a character:\n(enter x to exit)\n");
    while (a != `x')
    {
        a = getc(stdin);
        putchar(a);

        printf("\n");
    }
    printf("\nOut of the while loop. Bye!\n");
}
```

OUTPUT

```
Enter a character:
(enter x to exit)
i
H
x
x
Out of the while loop. Bye!
```

ANALYSIS

The char variable `a` is initialized before the the while Statement. In the while condition the relational expression `c != `x'` is tested. If the expression returns 1, which means the relation still holds, the statements under the while are executed. The looping continues as long as the expression returns 1 i.e returns True value. However when the user enters the character `x`, which makes the relational expression return 0, the looping stops.

3.3.1 The Infinite while Loop

You can also make a while loop infinite by putting 1 in the expression field, like this:

```
while (1) {
    statement1;
    statement2;
    .
    .
}
```

```

    .
}

```

Because the expression always returns 1, the statements inside the statement block will be executed over and over— that is, the while loop will continue forever. Of course, you can set certain conditions inside the while loop to break the infinite loop as soon as the conditions are met. The C language provides some statements, such as if and break statements, that you can use to set conditions and break the infinite while loop if needed. Details on the if and break statements are covered in Hour 10, "Getting Controls."

3.4 Do-while Loop

The while and for loops test the termination condition at the beginning of the loop. By contrast, the do-while loop, tests at the bottom *after* making each pass through the loop body; the body is always executed at least once. The syntax of the do is

```

do
    statement
while (expression);

```

First the *statement* is executed under the *do* clause, then expression in the while *is* evaluated. If it is true, *statement* is executed and again the while *is* evaluated and this looping process continues. When the expression becomes false, the loop terminates. The do-while is much less used than while and for. If the statement contains a set of statements, they must be enclosed in the braces .consider the example to understand the working of while-do loop.

- 1) Write a program to illustrate the working of do-while loop
/*working of Do-While loop*/

```

#include<stdio.h>
int main(void)
{
    int pass-word= 15;
    int code;
    int i=3;
    do
    {
        printf("Type the password to enter.\n");
        scanf("%d", &code);
        if (code==15)
            printf("Right code. You can Enter\n");
    }
}

```

```

        else
            ("Wrong code. No Entry\n");
            i=i-1;
        }
    while(i!=0)

}

```

Output:

```

13
Wrong code. No Entry
15
Right code. You can Enter

```

Analysis:

Here do part is executed to know the password. If it is right, entry is given otherwise not. Loop will repeat only for three set of inputs. For the fourth set variable i value does not satisfy the while condition. Hence loop is quit.

3.4.1 Limitation of Do-while Loop

In Do-while, the loop the is executed as least once before checking the validity condition. In Do-while structure entry to the loop is automatic (even with invalid input). Choice is given only for the continuation of the loop. Such problems are deemed to be *lack of control*. Example for such problem is given below.

- 1) Write a program to illustuate the limitation of Do-While loop.

```

/* program to reverse a number entered by the user */
#include <stdio.h>
main()
{
    int value, r_digit;
    printf("Enter the number to be reversed:\n");
    scanf("%d", &value);
    do {
        r_digit = value % 10;
        printf("%d", r_digit);
        value = value / 10;
    }
    while( value != 0 );
    printf("\n");
}

```

Output:

```

Enter the number to be reversed: 132

```

231

Enter the number to be reversed: 0

0

Analysis:

The above program reverses a number that is entered by the user. It does this by using the modulus % operator to extract the right most digit into the variable *r_digit*. The original number is then divided by 10, and the operation repeated whilst the number is not equal to 0. If user enters the number other than zero, program works. If user enters 0 (zero), then also the do part is executed, violating the condition at while. This means its possible to enter a *do { } while* loop with invalid data. Hence it is better to avoid the usage of Do-while statement. Its easy to avoid the use of Do-while construct by replacing it with other constructs. The above program can be written without Do-while construct. This is shown below.

```
/* rewritten code to remove construct */
#include <stdio.h>
main()
{
    int value, r_digit;
    value = 0;
    while( value <= 0 ) {
        printf("Enter the number to be reversed.\n");
        scanf("%d", &value);
        if( value <= 0 )
            printf("The number must be positive\n");
    }

    while( value != 0 )
    {
        r_digit = value % 10;
        printf("%d", r_digit);
        value = value / 10;
    }
    printf("\n");
}
```

Sample Program Output

Enter the number to be reversed.

-43

The number must be positive

Enter the number to be reversed.

423

324

3.5 Break Statement

The break statement is used to terminate loops or to exit from a switch. It is used within loop structures namely for, while, do-while and switch statement.

The syntax of break statement is

```
break;
```

The break statement is without any embedded expressions or statements. The **break** statement is frequently used to terminate the processing of a particular case within a **switch** statement. Usage of break within switch is already explained in the unit 2 of Module 1.

3.5.1 Break Statement Within Loops

statement following the loop. A break within a loop is generally protected within an if statement. This provides control towards the exit condition. This depicted in the below example.

Example : The following examples illustrate the use of the break statement in loops.

```
1) while (<expression>) {
    <statement>
    <statement>
    if (<condition which can only be evaluated here>)
        break;
    <statement>
    <statement>
}
```

// control jumps down here on the break

2) Write a program to illustrate the functioning of break statement inside a for loop.

/* a program to illustrate the functioning of break statement inside a for loop. */

```
#include <stdio.h>
int main()
{
    int i;
    for (i = 1; i < 10; i++)
    {
        printf_s("%d\n", i);
        if (i == 4)
            break;
    }
}
```

Output

1
2
3
4

Analysis:

Here break statement is given with respect to the if condition. When variable i value gets equal to four, loop gets terminated because of break statement.

3) Write a program to illustrate break statement within infinite for loop.
/* program to illustrate break statement within infinite for loop */

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

    for ( ; ; )
    {
        scanf("%d" , &t);
        if ( t==10 )
            break ;
    }
    printf("End of an infinite loop...\n");
}
```

5) Write a program to illustrate break statement within infinite while loop.
/* Program to Break an infinite while loop.*/

```
#include <stdio.h>

main()
{
    int c;

    printf("Enter a character: \n(enter x to exit)\n");
    while (1)
    {
        c = getc(stdin);
        if (c == `x')
            break;
    }
    printf("Break the infinite while loop. Bye!\n");
}
```

OUTPUT

Enter a character:


```

(enter x to exit)
H
I
x
Break the infinite while loop. Bye!

```

3.5.2 Can break Statement be Used Inside if Construct

The break does not work with if. It only works only with in loops and switches. After observing the sample programs where break is used in conjunction with if structure inside the for loop and while loop, the common wrong assumption is that the break statement can be used inside if construct. Thinking that a break refers to an if construct when it really refers to the enclosing while or for loop has created some high quality bugs. This is illustrated in the example below.

```

#include <stdio.h>
int main()
{
    int i;
    printf("Enter your choice from 1-3, where apple-
1,Orange-2, grapes- 3,    Enter 4 to exit ");
    scanf("%d",&i);
    if(i==1)
    Printf("Your choice of fruit is Apple ");
    if(i==2)
    Printf("Your choice of fruit is Orange");
    if(i==3)
    Printf("Your choice of fruit is Grapes");
    if (i == 4)
        break;
    }
}
/* This program reports error because break is used with if construct */

```

3.6 Continue Statement

Instead of breaking a loop, a need to stay in the loop but skip over some statements within the loop may occur. To do this, the continue statement is used. The continue statement takes the control to the beginning of the loop bypassing the statements inside the loop which have not yet been executed. The [continue](#) statement can only be used within the body of a [while](#), [do](#), or [for](#) statement. For better control over the program, like a break statement, continue is generally used in conjunction with if structure within loops. When continue is used the next iteration of a do, for, or while statement is determined as follows:

- In a while loop, jump to the test statement.
- In a do while loop, jump to the test statement.
- In a for loop, jump to the test, and perform the iteration. Then the compiler reevaluates the conditional expression and, depending on the result, either terminates or iterates the statement body

One good use of continue is to restart a statement sequence when an error occurs. Using the continue statement, as well as the break statement, may makes program hard to debug. Hence it is seldom used. The format of using continue in while loop and some examples of continue statement is given below.

```
while (<expression>)
{
...
if (<condition>)
continue;
...
// control jumps here on the continue
}
```

3) Write a program to find the sum Using the continue statement in for loop.

```
/* Using the continue statement */
#include <stdio.h>
main()
{
int i, sum;
sum = 0;
for (i=1; i<8; i++)
{
if ((i==3) || (i==5))
continue;
sum += i;
}
printf("The sum of 1, 2, 4, 6, and 7 is: %d\n", sum);
}
```

OUTPUT

The sum of 1, 2, 4, 6, and 7 is: 20

ANALYSIS:

Program is to calculate the sum of the integer values of 1, 2, 4, 6, and 7. because the integers are almost consecutive, a for loop is built in lines 9_13. The statement sum += i; sums all integers from 1 to 7 except for 3 and 5,.3nad 5 are skipped in the for loop by using continue statement.

This is done by evaluating the expression `(i==3) || (i==5)` in the if statement. If the expression returns 1 (that is, the value of `i` is equal to either 3 or 5), the continue statement is executed, which causes the sum operation to be skipped, and another iteration to be started at the beginning of the for loop. In this way, the sum of the integer values of 1, 2, 4, 6, and 7 is obtained, but 3 and 5 are skipped, automatically by using continue statement in for loop. After the for loop, the value of sum, 20, is displayed on the screen by the `printf()` function .

3) Write a program to find the odd numbers using continue statement in while loop.

`/* a program to find the odd numbers using continue statement*/`

```
#include <stdio.h>
main ()
{
    int x;
    x = 0;
    while (x < 10)
    {
        ++x;
        if (x % 2 == 0)
        {
            continue;
        }
        printf ("%i is an odd number.\n", x);
    }
}
```

Output

```
1 is an odd number.
3 is an odd number.
5 is an odd number.
7 is an odd number.
9 is an odd number.
```

3.7 Goto and Labels

C provides the infinitely-abusable goto statement, and labels to branch to. Formally, the goto statement is never necessary, and in practice it is almost always easy to write code without it. A *goto statement* causes an unconditional branching of the program to the statement associated with the label specified on the goto statement. A label has the same form as a variable name, and is followed by a colon. The scope of a label is the entire function. Label name must be unique for that program. Any number of goto labels can be used inside the program, provided each one of them is distinct. Format of goto statement is given below.

```

    for ( ... )
    statement
    for ( ... )
    {
        ...
        if (disaster)
        goto error; /* Here semicolon is associated with the label
error*/
    }
error: Statemnt; /* Here colon(:) is associated with the label error*/

```

Analysis:

Here error is a label name that tells the goto statement where to jump. Label name is placed in two places: One is at the place where the goto statement is going to jump (note that a colon must follow the label name), and the other is the place following the goto keyword. The rules to name a variable also applies to label also. Also, the place for the goto statement to jump can appear either before or after the statement. They are called forward goto or backward goto statements respectively.

Goto statement must be used in seldom. The programs written with goto are unreliable and hard to debug. The goto Statements are practically not necessary, and it is almost always easy to write code without it. However a goto may be necessary for exiting a loop from within a deeply nested loop.

Such as breaking out of two or more loops at once. since it can only The break statement be used from the inner most loop only, it cannot be used instead of goto in nested structure. Functioning of the goto statement is given in the below example.

1) Write a program to understand the functioning of the goto statement.

```

#include <stdio.h>
int main()
{
    int i, j;

    for ( i = 0; i < 10; i++ )
    {
        printf( "Outer loop executing. i = %d\n", i );
        for ( j = 1; j < 3; j++ )
        {
            printf( " Inner loop executing. j = %d\n", j );
            if ( i == 2 )
                goto stop;
        }
    }
}

```

```

    }
}
printf( "Loop is quit. i = %d\n", i );

stop: printf( "Jumped to stop. i = %d\n", i );
}

```

Output:

```

Outer loop executing. i =0
Inner loop executing. j =1
Outer loop executing. i =1
Inner loop executing. j =2
Outer loop executing. i =2
Inner loop executing. j =1
Jumped to stop. i=2

```

In this example, a goto statement transfers control to the point labeled stop when i equals 2.

3.8 Questions

- 1) Explain the working of for loop with example.
- 2) Explain the variations in for loop.
- 3) Explain the working of do – while loop with example.
- 4) Differentiate between do-while and while loop.
- 5) Explain goto, break and continue statements

3.9 Programing Exercise

- 1) Using while loop write a program to print natural numbers up to a given number.
- 2) Using while loop write a program to ask the user to enter a series of marks of a student. If the user enters –1, come out of the loop and print the average mark.
- 3) Using while loop write a program to ask the user for a number. Print the square of the number and ask a confirmation from the user whether the user want to continue or not.
- 4) Using do while loop Write a program to print the sum of digit of a given number
- 5) Using for loop write a program to print the odd numbers within a given number.
- 6) Using for loop Write a program to add the even numbers within a given number.

MODULE 3

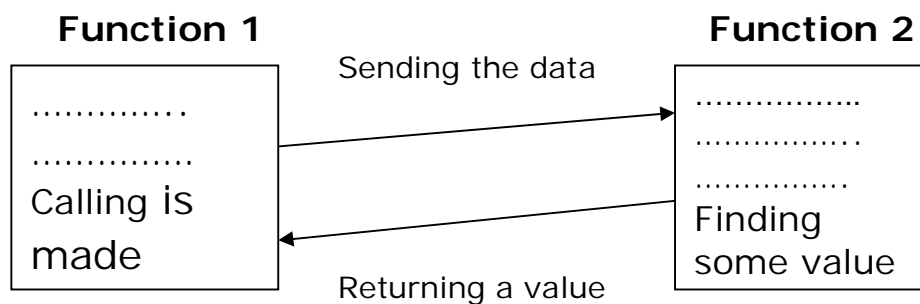
Unit 1 : Functions

1.1 Introduction

In C a large program can be divided into a number of smaller, complete and independent subprograms. These subprograms possess self-contained components, each of which has some unique, identifiable purpose. This task is called modularization and each sub program is called a module or a **function**. In other words a function is a complete and independent subprogram that can be developed and tested successfully. Functions are used (or invoked) by the main program or other subprograms. Thus a C program can be **modularized** through the intelligent use of such functions. This kind of approach to program development is called modular approach. (C does not support other forms of modular program development, such as the procedures in Pascal or the subroutines in FORTRAN.)

1.3 What is a Function?

The function takes a data from main () function and returns a value. To invoke a function call is made in the main () function. The function which sends the data to the function is called as the *Calling Function* and the function which is called by the calling function is called as the *Called Function*. This is described as given below.



Function 1 is the Calling Function 2 is the Called Function

To understand what a function is consider the program given below.

```
#include <stdio.h>
int main (void)
{
    printf ("Programming is interesting.\n");
    return 0;
}
```

This program prints the message "Programming is interesting" at the output terminal. The same program can be written using the function concept as given below.

```
#include <stdio.h>
void printMessage (void)
{
    printf ("Programming is interesting.\n");
}
int main (void)
{
    printMessage ();
    return 0;
}
```

Here the function called print Message () is used to print the message. More about functions is dealt in the following sections.

1.2.1 Functions are Used in C for the Following Reasons

1. Many programs require that a particular group of instructions be accessed repeatedly, from several different places within the program. The repeated instructions can be placed within a single function, which can then be accessed whenever it is needed. Thus the use of a function avoids the need for redundant (repeated) programming of the same instructions.
2. A different set of data can be transferred to the function each time it is accessed.
3. Logical clarity is achieved by using functions in Programs.
4. Testing and correcting errors is easy because errors are localized.
5. The flow of program and its code are easily understandable since the readability is enhanced while using the functions.
6. A single function written in a program can also be used in other programs also.
7. A function also promotes portability since programs can be written that are independent of system-dependent features.

1.2.2 Classification

C functions can be classified into two categories, namely library functions and user-defined functions. The library functions are standard functions available within C-Language library (built in functions) but user defined function are functions that are created by the user. User defined functions (UDF) usually use library functions to get the job done. Example of library functions are printf and scanf and main is an example of user defined function. This chapter deals in detail with user defined functions.

1.3 Structure of Function

The structure and usage of functions can be well understood by dividing it into three sections namely:

1. Function definition
2. Function declaration
3. Function invocation

Detailed explanation of each section is given below.

1.3.1 Function Definition

The function definition is the C code that implements what the function does. It contains two parts. The first part is function header and second part is function body. Function definition has the following syntax

data type function name (arguments list)

```

{
  function header      local declarations;
                       function statements;
                       return statement;
}
  
```

Consider the following example to understand the function definition in detail.

```

int sum(int x, int y)    //Function header
{
    int ans = 0;         //holds the answer that will be returned
    ans = x+y;           //calculate the sum
    return ans           //return the answer
}
  
```

In the first line of the above code function header `int sum (int x, int y)` It has four main parts

1. The name of the function i.e. sum

2. The parameters int x, int y, of the function enclosed in parenthesis
3. Return value type i.e. int
4. Function Body

Whatever is written with in { } in the above example is the body of the function

Analysis:

Function header consists of three parts namely,

Function Header

❶Data type ❷function_name (❸ argument1, argument2,..)

- 1) Data type of the function
- 2) Function name
- 3) Arguments list

- 1) Data type

The data *type* in the function header tells the type of the value returned by the function. This is also called as *return type*. Any of the basic data type such as int, float, char etc may appear in the data type part of the function. The data types are assumed to be of type **int** if they are not shown explicitly. When a function is not returning any value it may be declared as type void. For example,

(i) Void function name(...)

```
int function name(...)
float function name(...)
char function name(...)
function name(...)
```

(ii) Consider a function which just shows a message on the screen. In this function there is no need to return any value to the calling program. The data type of such function is void. This is a special specifier that indicates absence of data type.

```
/* void function example*/
#include <stdio.h>
void printmessage ()
{
    Printf( "I am learning functions in c");
}
int main ()
{
    printmessage ();
    return 0;
}
```

2) Function name

The function name can be anything. Normally it is named relevant to the function operation, as it is easy to keep the track of functions. The rules to form the function name are same as the rules of variables. For example,

```
counter();
square();
message();
output();
```

3) Arguments List

The arguments are also called as parameters. This tells what arguments the function needs when it is called (and what their types are). More than one argument is called arguments list. The arguments are optional. Each parameter consists of a data type specifier followed by an identifier, like any regular variable declaration (for example: `int x`) and which acts within the function as a regular local variable. They allow to pass arguments to the function when it is called. The different parameters are separated by commas. Data type of the function parameters is not mandatory and it is based on function requirements. The structure of arguments is given below.

data type function name (argument1, argument2, ..argumentn,)
or
data type function name (datatype argument1,datatype
argument2...datatype argument n) For example,

```
int square(int x,int y);
void name(double a, char car);
float space(p,x,y);
```

`void` can also be used in the function's parameter list to explicitly specify that the functions to takes no actual parameters when it is called. For example, function `printmessage` could have been declared as:

```
void printmessage (void)
{
    Printf( "I am learning functions in c");
}
```

It is optional to specify `void` in the parameter list. For a function with no parameters, a parameter list can simply be left blank .For example

```
void printmessage ()
{
    Printf( "I am learning functions in c");
```

```
}
```

More about arguments are dealt in the following sections.

4) Function Body

Whatever is written within the curly braces { } below the function header is called the body of the function. It contains the declarations and statements necessary for performing the required task. The function body contains three parts:

1. Local variables declaration

It specifies the variables needed by the function locally.

2. Function Statements

That actually performs the task of the function.

3. The return statement

Generally the return statement is used at the end of the function body. The keyword `return` is used to terminate the function and return a value. The return statement may or may not include an expression. Syntax of the return statement:

```
return;
return (expression);
```

The return statement can be absent altogether from a function definition, but it is considered as poor programming practice. If a return statement is not used in the function, control simply reverts back to the calling portion of the program without returning any information. The presence of an empty return statement (without the accompanying expression) is recommended in such situations to clarify the logic. The return statement is generally followed by a data variable or constant value or an expression. Some examples are

```
return;
return 3;
return n;
return ++a;
return (a*b);
```

The data type of the *return expression* must match that of the declared function data type. This is shown in the below example.

```
float add numbers (float n1, float n2)
{
```

```

        return n1 + n2; /*legal*/
        return 6; /*illegal, not the same data type*/
        return 6.0; /*legal*/
    }

```

1.3.2 Function Prototypes

All identifiers in C need to be declared before they are used. This is true for functions as well as variables. For functions the declaration needs to be done before the first call of the function. There are two ways to do this. They are

1. Define the function before it is called
2. Declare the function before it is called.

Both the types are explained the below sections.

1. Define the function before it is called

In this approach the function called is defined before the main () function. ie defining before main().Function definition comprises of function header and the function body . The function definition itself can act as an implicit function declaration. Hence the compiler knows what functions are used in main() before entering the main().Syntax is

```

Function definition[function header + function body]
    main()
    {
        function();
    }

```

For example consider a simple adding program which defines the function before it is called.

/* This program allows the user to input two numbers, then passes those two numbers into a sum function, and finally prints the returned sum.*/

```

#include <stdio.h>
int sum (int n1, int n2)      /* Function header*/
{
    int answer;               /*Function body */
    answer = n1 + n2;         /*Put the sum of the two numbers
into answer */
    return answer();          /* Exit function and return answer
*/
}

main ()
{

```

```

int number1,number2,total;
printf ("What is the first number? ");
scanf ("%d", &number1);
printf ("What is the second number? ");
scanf ("%d", &number2);

total = sum (number1, number2); /* Get the sum of the
two numbers from the return value of sum() */

printf ("The sum of %d plus %d is %d\n", number1,
number2, total);
}

```

Analysis:

- I. Here int sum (int n1, int n2) is a user defined function. That is, it is a custom function that programmer has created. It is defined before the main ().
- II. Function int sum() accepts two integer values, n1 and n2 from main() calculates the sum and returns an integer result using the return statement
- III. The function sum () has two parts one is function header, int sum (int n1, int n2) and other is function body. This header reveals that the function name is sum (), the arguments passed are int n1 and int n2 and it return data type is of int. The statements written within the curly braces are called function body.
- IV. Within the main function user is prompted for two numbers. Here the main function calls the sum function by passing the two numbers as arguments and then storing the return value into total. Functions such as scanf and printf, used in main() are called library functions. They exist in a pre-defined library of C (stdio.h)

2. Declare the function before it is called

In the above example the function is defined before the main program. For many programmers, this is a rather illogical way to work, as they want the main function of the program to be at the beginning, rather than user defined functions. If main function is defined before user defined function, compiler prompts error. Solution for this is to declare the function before it is used. Function declaration is also called function prototype, since they provide a model or blueprint for the function. Function declaration tells the compiler, "a function that looks like this is coming up later in the program", so it is like seeing reference of it before the function itself.

To **declare** a function prototype simply state the data type the function returns, the function name and in brackets list the type of parameters in the order they appear in the function definition. Function prototype contains the same information as the function header contains, but it ends with semicolon. The only difference between the header and the prototype is the semicolon (;). There must be the semicolon at the end of the prototype. It is a complete statement in itself.

The prototype of the function in the above example is

```
int sum (int n1, int n2);
```

Notice that function declaration does not contain the body of the function and function declaration is terminated with a semicolon. So a prototype shows what the inputs and output of the function is going to be, but doesn't show how the function does its work. It is basically a copy of the function without anything inside the curly brackets. For example consider the simple program with a function prototype

```
#include <stdio.h>
int sum (int, int);          /* Function prototype*/
int main (void)
{
    int total;
    total = sum (2, 3);      /* Function call*/
    printf ("Total is %d\n", total);
    return 0;
}
/* Function definition*/
int sum (int a, int b)      /* Function header*/
{
    return a + b;          /* Function body*/
}
```

Advantage of function prototype:

C compiler checks there turn data types and types and counts of all parameter lists. Try compiling the following:

```
#include <stdio.h>
int add (int,int); /* function prototype for add */
void main()
{
    printf("%d\n",add(3));
}
int add(int i, int j)
{
    return i+j;
}
```

The prototype causes the compiler to flag an error on the **printf** statement. Because the function call is having only one parameter while prototype is having two parameters.

1.3.3 Function Invocation

The function is called (or invoked) from main (). To invoke a function in main, the function name is written, followed by parentheses. The syntax of the function call is very similar to that of declaration, except that the return data type is not used. A semicolon is placed at the end of the call expression. The following facts occur when function is invoked.

- When a function is invoked, **control is transferred to the first the statement in the functions body**. Computer immediately begins executing statements from the beginning of the called function. Each time the function is called; execution always starts at the beginning of the function.
- Execution continues inside the called function until either:
 - It reaches the right } at the end of the function
 - Or a **return** statement.
- Either way, the function stops at this point and execution picks up right where it had left off in the original function (e.g., back in **main**)
-

If function is returning a value, *variable* can be assigned to the return-value of the function. For example,

```
#include <stdio.h>
int addition (int a, int b)
{
    int r;
    r=a+b;
    return (r);
}
int main ()
{
    int z;
    z = addition (5,3);
    Printf("The result is%d ",z);
    return 0;
}
```

Analysis:

1. Here function addition is defined before the main. Hence there is no need of function prototype.

2. Function addition adds two arguments passed by main and returns the result to the variable z in the main function.
3. Instead of void main int main is used. Return zero is used at the end of the program in main just to say that program is returning int value zero to main. void main() is also used to avoid the function of return because void itself specifies that there is no value returned to main

Returning values from functions

When a function completes its execution, it can return a single value to the calling program. By default called function returns an int value to main. Usually this return value consists of an answer to the problem the function has solved. If user is desired that a function should return a value other than an int, then it is necessary to explicitly mention the the data type in the calling functions as well as in the called function. For e.g

```

square (float X)
{
float y;
Y = x * x;
return (y);
}
main ( )
{
float a,b,
printf ("\n Enter any number: ");

scanf ("% f", &a );
b = square (a)
printf ("\n square of entered number % f is % f", a,b);
}

```

Result:

```

Enter any number: 2.5
square of entered number 2.5 is 6.000000

```

Analysis:

Square of 2.5 is not 6.000. Why the result is different? This is because the function square(float x) by default, always returns an integer value. The fractional part is truncated. In order to return the float type, float data type must be mentioned at the function as shown in the below example.

```

main ( )
{
float square ( );

```



```

float a, b;
printf ("\n Enter any number ");
scanf ("%f" &a);
b = square (a);
printf ("\n square of entered number  % f is % f, " a,    b);
}
float square (float x)
{
float y;
y= x *x;
return ( y);
}

```

Result:

Enter any number:2.5

square of entered number 2.5 is 6. 2500000

Some facts about functions

The following points must be noted about functions

- (i) C program is a collection of one or more functions.
- (ii) A function gets called when the function name is followed by a semicolon. Rules for declaring the functions name are same as that of variables declaration rules. for e.g.

```

main ( )
{
    {
        int Message1 ( );
    }
}

```

Here message 1(); is the function name

- (iii) A function is defined when function name is followed by a pair of braces in which one or more statements may be present for e.g.

```

int message1 ( )
{
    statement 1;
    statement2;
    statement 3;
}

```

- (iv) Any function can be called from any other function even main () can be called from other functions. for e.g.

```
main ( )
{
    message ( );
}
int message ( )
{
    printf ( " \n Hello");
    main ( );
    return 0
}
```

(v) A function can be called any number of times for eg.

```
message ( )
{
    printf ("\n Hello");
}
main ()
{
    message ( );
    message ( );
}
```

(vi) The order in which the functions are defined in a program and the order in which they get called need not necessarily be same for e.g.

```
void message2 ( )
{
    printf ("\n I am learning functions in C");
}
void message1 ( )
{
    printf ( "\n Hello ");
}
main ( );
{
    message1 ( );
    message2 ( );
}
```

(vii) A function can call itself such a process as called 'recursion'.

(viii) A function can be called from other function, but a function cannot be defined in another function. Thus the following program code would be wrong, since function say() is being defined inside another function main ().

```

main ( )
{
    printf ("\nThis section is in main");
    void say( )
    {
        printf {"\nFunctions are easy."};
    }
}

```

(ix) Any C program contains at least one function

(x) If a program contains only one function, it must be main ()

(xi) In a C program if there are more than one functions are present then one of these functions must be main () because program execution always begins with main ().

(xii) There is no limit on the number of functions that might be present in a C program

(Xiii) Each function in a program is called in the sequence specified by the function calls in main ()

1.4 Types of Arguments

An argument is an entity used to pass the data from calling function to the called function. The arguments are also called as the parameters. There are two types of parameters. They are

i) **Actual Parameters:** The arguments or parameters used in the calling function to call the called function are called as the Actual parameters.

ii) **Formal parameters:** The arguments or parameters used in the called function header are called as the Formal parameters. Formal arguments are the arguments available in the function definition. For example consider the below example.

```

main()
{
    function1(a1,a2,a3.....an)
}

function1(f1,f2,f3.....fn);
{
    functionbody;
}

```

here a1,a2,a3 are actual arguments and f1,f2,f3 are formal arguments.

```

Void add (int a, int b)           / *Formal parameters */
{

```

```

        int c;
        c=a+b;
        printf ("The sum is %d",c);
        return;
    }
    main()
    {
        int x,y;
        Printf(" Enter two integers")
        scanf("%d %d" ,&x, &y);
        add (x, y);          / *Actual parameters */
    }

```

Analysis:

1) Consider the above program. The function definition begins with this declaration:

```
add (int a, int b)
```

This line informs the compiler that add() uses two arguments called a and b, both a, b variables are of the type int. Both the a and b variables are called *formal arguments*. Like variables defined inside the function, formal arguments are local variables, private to the function. That means programmer doesn't have to worry about duplicating variable names used in other functions. These variables will be assigned values each time the function is called.

2) To declare formal arguments, each variable must be preceded by its data type individually. That is, unlike the case with regular declarations, a list of variables of the same type cannot be used. For example,

```

void sum (int x, y, z)      /* invalid function header */
void sum (int x, int y, int z) /* valid function header */

void add (int a, int b)     /* valid function header */
void add (int a, b)        /* Invalid function header */

```

3) Another way of declaring formal arguments is

```

void add (a, b)
int a,b;

```

4) In the above program int a and int b values are assigned by actual arguments in the function call. Consider the line in main program add (x,y);

The actual arguments are the x and y. These values are assigned to the corresponding formal arguments in add() — for the variables x and y respectively.

5) The actual argument can be a constant, a variable, or an even more elaborate expression. Regardless of which it is, the actual argument is evaluated, and its value copied to the corresponding formal argument for the function.

6) Instead of using different variable names int a ,int b in the called function, the same variable name as in the caller function i.e int x, int y can be used. But still compiler treats them as different variables only because they are in called functions.

7) Both the arguments actual and formal should match in number, type and order. The values of actual arguments are assigned to formal arguments on a one to one basis starting with the first argument as shown below. If not compiler will flag error. For example,

```
void fun(int x, int y, char c1,char c2);
main()
{
    int x, y;
    char c1,c2;
    -----
    -----
    un(x,y,c1,c2)
}
void fun(x,y,c1,c2)
{
    char x,y;
    int c1,c2; /* Data type mismatch*/
    .....
    .....
}
```

6) When a function call is made only a copy of the values actual arguments is passed to the called function. What occurs inside the functions will have no effect on the variables used in the actual argument list.

1.5 Types of Functions

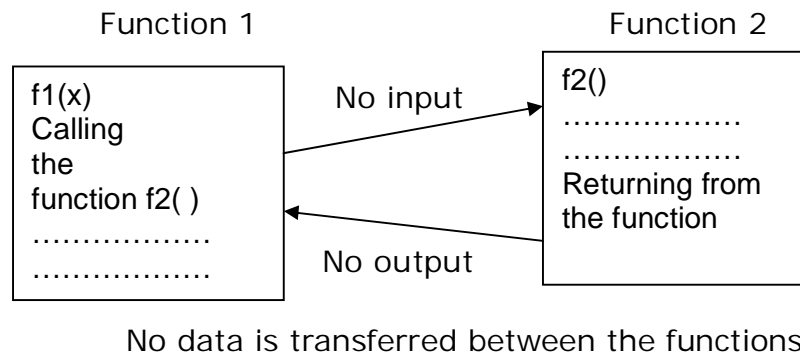
According the arguments and the returning value, functions are divided into three categories.

- 1) A function with no arguments and no return value
- 2) A function with no arguments and return a value

- 3) A function with an argument or arguments and returning no value
- 4) A function with arguments and returning a value.

1.5.1 A Function With No Arguments And No Return Value

If a called function does not have any arguments, it is not able to get any value from the calling function. Also if it does not return any value, the called function is not receiving any value from the function when it is called. That is, there is no data transfer between the calling function and the called function. This can be represented as given below.



Example :

```
main( )
{
    message ( );
}

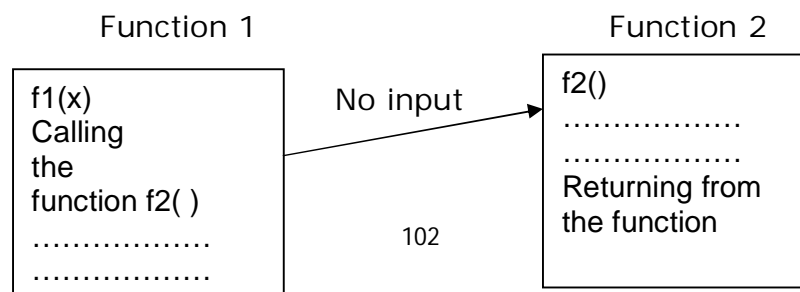
void message( )
{
    printf("\n WELCOME");
}
```

Analysis:

main () is the calling function and message () is the called function. Between the functions, no data is transferred.

1.5.2 A Function with No Arguments and Returns a Value

A function which does not get value from the calling function but it can return a value to calling program. This can be represented as given below.





Example :

```
main( )
{
    int a;
    a=todaytemp( );
    printf("\n The returned value form the todaytemp( )
    function is %d",a);
}

todaytemp( )
{
    return(100);
}
```

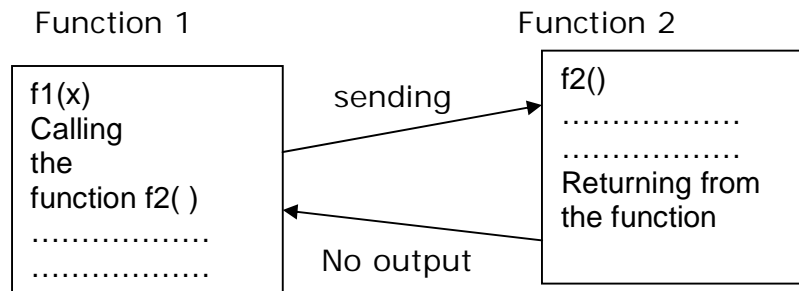
Analysis:

Here the called function does not receive any parameters from main ie called function but returns a value to called program.

1.5.3 A Function with Arguments and Returns No Value

A function has an argument or set of arguments. Through arguments calling function can pas values to function called. But calling function does not receive any value.

That is, data is transferred from calling function to the called function but no data is transferred from the called function to the calling function. A function that does not return any value cannot be used in an expression it can be used only as independent statement. This is represented as given below.



Example :

/*Program to find the largest of two numbers using function*/

```
#include
main()
{
    int a,b;
    printf("Enter the two numbers");
    scanf("%d%d",&a,&b);
    largest(a,b)
}
/*Function to find the largest of two numbers*/
largest(int a, int b)
{
    if(a>b)
        printf("Largest element=%d",a);
    else
        printf("Largest element=%d",b);
}
```

Analysis:

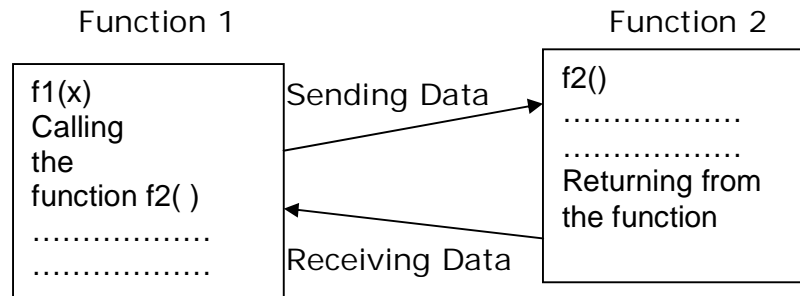
In the above program the calling function reads the data and passes it on to the called function. But called function does not return any value.

1.5.4 A Function with Arguments and Returning a Value

Here arguments are passed by caller function to called function. Called function returns value to caller function. Called function can receive any number of arguments but can *return only one result*. If multiple results are required multiple functions must be used. For example,

```
fun (a,b)
int a,b;
{
    int y,z;
    y=a+b;
    z=a-b;
    return y;
    return z; /* Two return statements for one single
function called is invalid */
}
```

A function with arguments and returning a value can be represented as given below.



No data is transferred between the functions

Example :

```

#include<stdio.h>
float findaverage(float a, float b); /*Function prototype*/
main()
{
    float a=5,b=15,result;
    result=findaverage(a,b);
    printf("average=%f \n",result); /*Function prototype*/
}
float findaverage(float a, float b)
{
    float average;
    average=(a+b)/2;
    return(average);
}
    
```

1.6 Questions

- 1) What is a function?
- 2) Explain the structure of a function.
- 3) What is the difference between function declaration and function definition?
- 4) Explain function invocation with explain with an example.
- 5) Explain the function types with examples.

1.7 Programing Excersice

- 1) Write a program to print "Hello World" in the main function and "Welcome To C" in another function.
- 2) Write a program to add three given numbers using function.
- 3) Write a program to calculate the tax of n given number of employees. Use a separate function to calculate the tax. Tax is 20% of basic if basic is less than 9000 otherwise tax is 25% of basic.
- 4) Write a program to check a number is prime or not using function.
- 5) Create a function add() which will calculate the sum of the array elements passed as parameters to it
- 6) Write a function to calculate the factorial of the number.
- 7) Create a user defined function to find the minimum value of three float numbers which are passed as the arguments to it.
- 8) Write a function range() which will take value of A and B as parameter and displays total numbers divisible by 3 in the range A to B.
- 9) Write a function which will take a year as argument and find whether that year is a leap year or not?

MODULE 3

Unit 2: LOCAL AND GLOBAL VARIABLES

2.1 Local variables

A local variable exists inside the specific function that creates them. They are unknown to other functions. Local variables cease to exist once the function that created them is completed. They are recreated each time a function is executed or called. Local variables must always be defined at the top of a block i.e at the starting of the brace ({). When a local variable is defined - it is not initialized by the system and must be initialized. For example consider the below program,

```
main()
{
    int x;
    float y;
    .....
    tree();
}

void tree()
{
    int x;
    float y;
    .....
}
```

Here each x and y variables are LOCAL to its own routine and are called local variables. The x and y variables in main () is unrelated to the x and y variables in the tree function. (Local variables are also called "automatic".)

2.2 Global Variables

Global variables are created by declaring them outside any (including main) function. These variables can be accessed by any function throughout the entire program. They do not get recreated if the function is recalled. Global variables are initialized to zero value by the system when they are declared. To understand global variable consider below example,

```
int max;
main()
{
    Printf(" The value of global variable max is: %d",max);
    max=max+1;
}

f1()
{
```

```

        max++;
        Printf(" The value of global variable max in the fuction is:
%d",max);
        ....
    }

```

Output:

The value of global variable max is: 0

The value of global variable max in the function is: 2

Analysis:

Here int max is global variable. The int max can be used in both main and function f1 and any changes made to it will remain consistent for both functions. When the global variable is declared its value is initialized to zero by the system. this is seen by the first output. the value of variable is incremented by one in the main() function. In the function f1, max is called. Even after exiting the main() the value of max variable is retained ie 1 and it is incremented by one in the functionf1. The value of variable is two. This is seen in the second output.

To understand the difference between local and global variable consider the below example.

```

int i=4;                                /* Global declaration */
main()
{
    i++;                                /* global variable */
    func();
    .....
}

func()
{
    int i=10;                            /* local declaration */
    i++;
    .....                                /* local variable */
}

```

Analysis:

The variable i in main is global and will be incremented to 5. The variable i in func is internal and will be incremented to 11. When control returns to main the internal variable will die and any reference to i will be to the global.

2.3 Call by Value and Call by Reference

Functions communicate with each other by passing the arguments. There are two ways of passing the arguments namely, call by value and call by reference.

2.3.1 Call by Value

In the preceding examples it is seen that whenever a function is called, the values of variables are passed to the called function from calling function. Such function calls are called "calls by value".

In this *call by value* method the value of each of the actual arguments in the calling function is copied into corresponding formal arguments of the called function. With this method the changes made to the formal arguments in the called function have no effect on the values of actual argument in the calling function. The following program illustrates this:

```
main ( )
{
    int a = 10, b=20;
    swap (a,b);
    printf ("\na = % d, b = % d", a,b);
}

swap (int x, int y)
{
    int t;
    t = x;
    x = y;
    y = t;
    printf ( "\n x = % d, y = % d" , x, y);
}
```

output :

x = 20, y = 10

a =10, b =20

Analysis:

In the above program the initial values of variables are a=10 and b=20. These variables are passed as arguments to called function swap(). Function swap() has formal arguments x and y equivalent of a and b variables and this function swaps the variables which is seen in the second line of output. After swapping the control returns to main(), but the values of variables are unaltered in the main() and this is seen in first

line of output. This is because when the arguments are passed their copy is only sent not the original arguments.

2.3.2 Call by Reference

In the second method the addresses of actual arguments in the calling function are copied in to formal arguments of the called function. This means that using these addresses the actual arguments can be accessed and hence programmer would be able to manipulate them. Since call by reference uses pointers it is explained in the chapter ... Pointers.

2.4 Recursion

Recursion is the process in which a function repeatedly calls itself to perform calculations. They are commonly used in applications in which the solution to a problem can be expressed in terms of successively applying the same solution to subsets of the problem. Typical applications are games and sorting trees and lists. Recursive algorithms are not mandatory, usually an iterative approach can be found. Recursive functions are most commonly illustrated by an example that calculates the factorial of a number. The factorial of a positive integer n , written $n!$, is simply the product of the successive integers 1 through n . The factorial of 0 is a special case and is defined equal to 1. So $5!$ is calculated as follows:

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

To understand the recursive functions consider the program given below.

/* Program using recursive function to print the factorial of a given number*/

```
#include <stdio.h>
#include <conio.h>
unsigned long fact(int);
void main()
{
    unsigned long f;
    int n;
    clrscr();
    printf("\nENTER A NUMBER: ");
    scanf("%d",&n);
    f=fact(n);
    printf("\nFACTORIAL OF %d IS %ld",n,f);
    getch();
}
unsigned long fact(int a)
{
```

```

unsigned long fac;
if(a==1 || a==0)
return(1);
else
fac=a*fact(a-1);
return(fac);
}

```

Output:

ENTER A NUMBER:0
 FACTORIAL OF 0 IS 1

ENTER A NUMBER:3
 FACTORIAL OF 3 IS 6

Analysis:

The factorial function includes a call to itself make this function recursive. When the function is called to calculate the factorial of 3, the value of the formal parameter *a* is set to 3. Because this value is not zero, the following program statement

```

fac=a*fact(a-1);

```

is executed, which, given the value of *a*, is evaluated as

```

fac=3 * fact(2);

```

This expression specifies that the factorial function is to be called, this time to calculate the factorial of 2. Therefore, the multiplication of 3 by this value is left pending while factorial (2) is calculated. Even though the same function is called again, it should be conceptualized as a call to a separate function. Each time any function is called in C—be it recursive or not—the function gets its own set of local variables and formal parameters with which to work. Therefore, the local variable *fac* and the formal parameter *a* that exist when the factorial function is called to calculate the factorial of 3 are distinct from the variable *fac* and the parameter *a* when the function is called to calculate the factorial of 2.

2.5 Storage Class

Storage class is a concept in c which provides information about the variable's visibility, lifetime and scope. The meaning of each term is as follows.

Scope: Scope is the region in which a variable is available for use.

Visibility: The program's ability to access a variable from memory is called as visibility of variable.

Lifetime: The lifetime of the variable is duration of the time in which a variable exists in the memory during execution.

Apart from data type variables have storage class. In C , there are four types of storage classes. They are

1. Local or Automatic variables
2. Global or External variables
3. Static Variables
4. Register Variables

2.5.1 Local or Automatic Variables

Automatic variables are declared inside a particular function and they are created when the function is called and destroyed when the function exits. Automatic variables are local or private to a function in which they are defined. Other names of automatic variable are *internal variable* and *local variable*. A variable which is declared inside the function without using any storage class is assumed as the automatic variable because the default storage class is automatic.

Storage: Memory.

Default initial Value: Garbage.

Scope: Local (to block in which variable defined)

Lifetime: Till control remains within block where it is defined.

Syntax:

auto<variable>

Here auto is the keyword for automatic variable.

For example consider the below program.

```
main()
{
    auto int x, y;
    int a,b;
    x = 10;
    printf("Values : %d %d", x, y);
}
```

Analysis:

Here x and y are defined as automatic variables by the usage of keyword auto. Variables a and b are also considered as auto because any variable which does not have any storage class explicitly specified , it is considered as automatic storage class only.

2.5.2 Global or External Variables

External variable:

External variable is a global variable which is declared outside the function. The memory cell is created at the time of declaration statement is executed and is not destroyed when the flow comes out of the function to go to some other function. The global variables can be accessed by any function in the same program. A global variable can be declared externally in the global variable declaration section.

Specifications:

Storage: Memory.

Default initial Value: Zero

Scope: Global

Lifetime: As long as the program execution doesn't come to an end.

Syntax:

There is no keyword to state any variable as global. if any variable is declared outside the function without any other storage class , it is implicitly considered as global variable.

For example consider the program,

```
int x = 100;
main ( )
{
    ....
    x = 200;
    f1 ( );
    f2 ( ); ...
    ....
}
f1 ( )
{
    x = x + 1;
    ....
    ....
}
f2 ( )
{
    x = x + 100;
    ....
    ....
    Printf(" The final value of global variable x is %d",x );
}
```

Output;

The final value of global variable x is 307

Analysis:

The global variable x is declared above all the functions. It can be accessed by all the functions as main (), f1 () and f2 (). The initial Value assigned to it is 100. When it is accessed by main(), its value is 200. When f1() function accesses it , x with the last recent value 200 is passed and f1() function manipulates it to 201. Similarly when function f2() is executed the value of x becomes 307 which is given in output.

2.5.3 Static Variables

These variables are alive throughout the program. A static variable can be initialized only once at the time of declaration. The initialization part is executed only once and retains the value till the end of the program.

Storage: Memory.

Default initial Value: Zero.

Scope: Local (to block in which variable defined)

Lifetime: Value of the variable remains as it is between function calls.

Syntax:

A variable can be declared as static by using the keyword "static". For example

static int x ; For example consider the program,

```
main()
{
    int j;
    for(j=1; j<=3; j++)

        stat();
}
stat();
{
    static int x=0;
    x=x+1;
    printf("The Value of X is %d\n",x);
}
```

Output:

The Value of X is 1
The Value of X is 2
The Value of X is 3.

Analysis:

During the first call to `stat()` function, `x` is incremented to 1. Because `x` is static, this value persists and therefore the next call adds another 1 to `x` giving it a value of 2. The value of `x` becomes 3 when third call is made. If variable `x` would have declared as an auto then output would have been `x=1` all the three times.

2.5.4 Register Variables

A variable is usually stored in the memory but it is also possible to store a variable in the processor's register by defining it as register variable. The registers access is much faster than a memory access, keeping the frequently accessed variables in the register will make the execution of the program faster. Since only a few variables can be placed in a register, it is important to carefully select the variables for this purpose. However `c` will automatically convert register variables into normal variables once the limit is exceeded.

Syntax;

Register variables are declared by using the keyword "register " . For example,

```
register int x;
```

Specifications:

Storage: CPU Registers.

Default initial Value: Garbage.

Scope: Local (to block in which variable defined)

Lifetime: Till control remains within block where it is defined.

For example consider the program,

```
main ( )
{
    register x , y z;
    Printf(" Enter two numbers");
    scanf("%d %d",&x,&y);
    z=x+y;
    printf("\n The sum  is %d",z);
}
```

Output:

Enter two numbers: 2.4

The sum is 6

Analysis:

In the above program, all the variables are stored in the registers instead of memory.

Programs:

1) Write a program in C which incorporates a function using parameter passing and performs the addition of three numbers. The main section of the program is to print the result.

```
#include <stdio.h>
int calc_result( int var1, int var2, int var3 )
{
    int sum;
    sum = var1 + var2 + var3;
    return( sum ); /* return( var1 + var2 + var3 ); */
}

main()
{
    int numb1 = 2, numb2 = 3, numb3=4, answer=0;
    answer = calc_result( numb1, numb2, numb3 );
    printf("%d + %d + %d = %d\n", numb1, numb2, numb3,
answer);
}
```

2) Write a recursive function to print a given number in reverse order.

```
#include <stdio.h>
#include <conio.h>
int reverse(unsigned long);
void main()
{
    unsigned long num;
    clrscr();
    printf("\nENTER A NUMBER: ");
    scanf("%lu",&num);
    printf("\nREVERSE OF %lu IS ",num);
    reverse(num);
    getch();
}

int reverse(unsigned long n)
{
    int dig;
    if(n==0)
        return 1;
    else
    {
        dig=n%10;
        n=n/10;
        printf("%d",dig);
    }
}
```

```

        reverse(n);
    }
}

```

- 3) Write a recursive function to print the Fibonacci series up to a given number.

```

#include <stdio.h>
#include <conio.h>
unsigned long fib(int);
void main()
{
    int n,i;
    unsigned long f;
    clrscr();
    printf("\nENTER A NUMBER: ");
    scanf("%d",&n);
    printf("\nTHE FIBONNACI SERIES UPTO %d NUMBERS
IS:\n",n);
    for(i=0;i<n;i++)
    {
        f=fib(i);
        printf("%lu ",f);
    }
    getch();
}

unsigned long fib(int x)
{
    unsigned long res;
    if(x==0)
        return(0);
    else
        if(x==1)
            return(1);
    else
    {
        res=fib(x-1)+fib(x-2);
        return(res);
    }
}

```

- 4) Write a program to pass a constant as a variable.

```

# include <stdio.h >
void repchar(char,int);
int main()
{

```

```

char char;
int num;
printf("enter a character:");
scanf("%c", &char);
printf("enter a number of times to repeat it:");
scanf("%d", &num);
repchar (char,num);
return 0;
}
/*repchar () function definition*/
void repchar(char ch, int n)
{
for(int j=0,j<n;j++)
Printf("%c",ch);
Printf("/n");
}

```

2.6 Questions

1. Explain recursion with example
2. Define global and local variables. Explain with examples.
3. What is meant by storage class? Explain the four different storage classes with example
4. List the rules governing the return statement.

2.7 Programming Exercise

- 1) Write a function to calculate the factorial of the number
- 2) Create a user defined function to find the minimum value of three float numbers which are passed as the arguments to it.
- 3) Write a function which will take a year as argument and find whether that year is a leap year or not?
- 4) Write the output of following program. Take suitable input. 4

```

#include<stdio.h>
void large( )
{
int a,b;
printf("Enter values of a and b :");
scanf("%d %d",&a,&b);
if(a<b)
printf("large : %d",b);
else
printf("large : %d",a);
}

```

```
main( )
{
    large( );
    large( );
}
```

MODULE 4

UNIT 1: Arrays

1.1 Introduction

In the previous chapters how to declare a variable with a specified data type, such as char, int, float, or double is explained. In many cases, there is a need to declare a set of variables that have the same data type. For example, to store date of birth, three different variables

```
int day, month, year;
```

are defined and each of which can store a single whole number. If date of birth is 1-2-2009, this can be stored in the variables as, day = 1; month = 2; year = 2009; similarly, to store five different numbers, five variables would have to be declared such as,

```
int num1, num2, num3, num4, num5;
```

With five numbers this is fairly easy. But what if a hundred items has to be stored? Can hundred variables be declared? If yes, program would become rather large and repetitive and cumbersome. As a solution for such problems programming language C has given the concept of an array. In array a variable is allowed to store more than one item. All of the items in a single array variable must be the same type - e.g. an **integer** or a **float** or a **char**.

1.2 Defining an Array

An array is a collection of variables of the same data type accessed by indexing. Each item in an array is called an element. All elements in an array are referenced by the name of the array and are stored in a set of consecutive memory slots.

Arrays are classified as single dimensional and multidimensional arrays. Array containing single column elements are called single dimensional

array or one dimensional or linear array. More than one column are called multidimensional array.

1.2.1 Declaring Single Dimensional Array

Like any other variable arrays must be declared explicitly before they are used. The general form of declaration is:

```
data-type Array-Name[Array-Size];
```

Here data-type is the type specifier that indicates what data type the declared array is such as int, float double or char. Array-Name is the name of the declared array and rules of declaring the variable applies to array name. Array-Size defines how many elements the array contains. This is always an integer. Note that the brackets ([and]) are required in declaring an array. The bracket pair ([and]) is also called the array subscript operator.

For example, an array of integers is declared in the following statement,

```
int array_int[8];
```

where int specifies the data type of the array whose name is array_int. The size of the array is 8, which means that the array can store eight elements (that is, integers in this case).

The other examples are

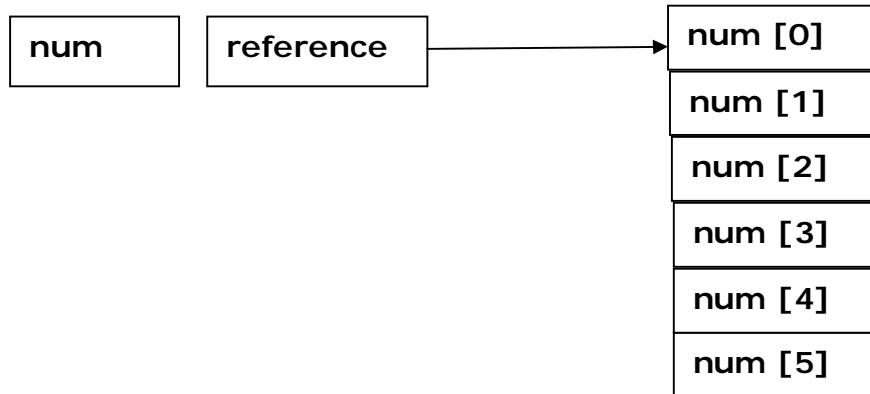
| | |
|-----------------|----------------------------|
| char C[10] | An array of characters |
| float plum[6] | An array of float numbers |
| double page[12] | An array of double numbers |

1.2.2 Array Index

An array is implemented as continuous storage, the index-based access. To access element n , in array, position or index of array is important the index of the first element (sometimes called the "*origin*") varies by language. There are three main implementations: *zero-based*, *one-based*, and *n-based* arrays, for which the first element has an index of zero, one, or a programmer-specified value respectively. The zero-based array is more natural and adopted by the C programming language.

For ex, consider the array declaration `int num[6];` This array consists of five elements and index of first element is 0 , second is one and so on. This described in the below figure.

int num[6];



1.3 Initialization of One Dimensional Array

There are two categories of array namely,

1. Static array
2. Dynamic array

Declaration for static array specifies the array size, which cannot be altered afterwards. But for dynamic arrays the size can be altered. Using dynamic memory the dynamic array's size can be modified. This is discussed pointers in chapters.

Once an array is declared, it must be initialized. Otherwise array will contain the garbage values. There are two different ways in which we can initialize the static array:

1. Compile time
2. Run time

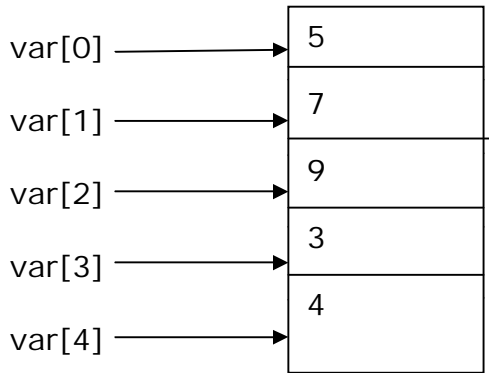
1.3.1 Compile Time Initialization

This initialization is done while writing the program itself. Following is the syntax of initialization of the array.

`data-type array-name[size] = { list of values separated by comma };`
0
]; For example:

int var[5] = {5, 7, 9, 3, 4};

Here, the array variables are initialized with the values given in front of it. That is, the 0th element of array will contain 5, 1st will contain 7 and so on. Remember, the array indexing or subscripting is always done from 0th position. The values of the array will be get stored in following manner



Each element in the array will be referred individually by its index such as, var[0], var[1], var[2], . . .

They can also be initialized individually such as:

```
var[0] = 5;
```

```
var[1] = 7;
```

```
var[2] = 9;
```

etc. after declaration of the array.

In the array declaration, the size can also be omitted. In such cases the compiler will automatically identify the total number of elements and size will be given to it. For example:

```
int max[ ] = {8, 6, 7, 4, 5, 3, 0, 1};
```

This array is not declared with size, but initialized with values. Here, size of the array will be automatically set to 8. Now it can be referred as the general array. Compile time initialization can also be done partially. Such as

```
float x[5] = {1.2, 6.9, 4.1};
```

```
x[0]=1.2, x[1]=6.9, x[2]=4.1, x[3]=0, x[4]=0
```

Here, array 'x' is having size 5, but only 0th, 1st, and 2nd elements are declared with values. The remaining 3rd and 4th element of the array will be

automatically set to 0. This property is applicable to all numerical type of the

array in C. However, if we have more values than the declared size, the compiler will give an error. For example –

```
int arr[5] = {56, 10, 30, 74, 56, 32};
```

This is the illegal statement in C.

```
int n[10] = {0};
```

First element of n[] is explicitly initialized to 0, and the remaining elements by default are initialized to 0.

```
int n[10];
```

In this all 10 elements contains junk values. The individual array element can be assigned to a variable with a statement such as

```
int G;  
G = grade [50];
```

The statement assigns the value stored in the 50th index of the array to the variable g. A value can be stored into an element in the array simply by specifying the array element on the left hand side of the equals sign. This is as shown in the statement,

```
grades [100]=95;
```

The value 95 is stored into the element number 100 of the grades array.

1.3.2 Run Time Initialization

An array can initialized at run time by the program or by taking the input from the keyboard. Generally, the large arrays are declared at run time in the program itself. Such as –

```
int sum[20];  
for (i = 0; i<20; i++)  
    sum[i] = 1;
```

Here, all the elements of the array 'sum' are initialized with the value one. The loop control structures are applicable in these cases. The scanf function can also be used to input the values from the user. In such cases, the loop control structure is applicable. For example:

```
int sum[10], x;  
printf("Enter 10 numbers: ");  
for(x=0; x<10; x++)  
    scanf("%d", &sum[x]);
```

Here, the array 'sum' is initialized by taking the values as input from

the keyboard.

1.4 Entering Data Into The Array

To enter the data into array, consider the below lines.

```
for (i=0; i<= 29; i++)
{
    printf ("\n Enter marks")
    scanf ("%d", &marks [i]);
}
```

The above section will read about 30 elements numbered from 0 to 29 in to the array named marks . This will take input from the user repeatedly 30 times.

1.5 Reading Data From Array

```
for ( i=0; i <= 29; i++);
Printf(The value entered into array marks [%d] is %d", i,marks[i]);
```

In order to understand the working of array , consider the below programs.

1)Write a program to initialize ten elements of an array.

```
/* Initializing an array */
```

```
#include <stdio.h>
main()
{
    int i;
    int list[10];
    for (i=0; i<10; i++)
    {
        list[i] = i + 1;
        printf( "list[%d] is initialized with %d.\n", i, list[i]);
    }
}
```

Output:

```
list [0] is initialized with 1.
list [1] is initialized with 2.
list [2] is initialized with 3.
list [3] is initialized with 4.
list [4] is initialized with 5.
list [5] is initialized with 6.
```

list [6] is initialized with 7.
 list [7] is initialized with 8.
 list [8] is initialized with 9.
 list [9] is initialized with 10.

ANALYSIS

An integer array, called list is initialized to ten elements. In order to enter the values into array in the run time, for loop is set to iterate for ten times. The statement `list[j] = i + 1;` assigns the values to array array. In the first iteration `i=0`, which is assigned as array index and the value is `i+1` ie `0+1=1` is assigned as the value for array index `list[0]`. This value is printed. For the next iteration index is 2 and the computed value is 2. This process continues till for loop condition is invalid.

1) Write a program to read 10 data from users and to find sum of them using array.

`/* Input 10 numbers from user and find the total of all of them */`

```
#include<stdio.h>
main( )
{
    int val[10], i, total=0;
    printf("Enter any ten numbers: ");
    for(i=0;i<10;i++)
        scanf("%d", &val[i]); // input numbers
    for(i=0;i<10;i++)
        total = total + val[i]; // find total
    printf("\nTotal is: %d", total);
}
```

Sample Output:

Enter any ten numbers: 1 2 3 4 5 6 7 8 9 10
 Total is: 55

1.6 The Size of an Array

An array is stored in consecutive memory locations. Given an array, like this:

`data-type Array-Name[Array-Size];`

it is possible to calculate the total bytes of the array by the following expression:

`sizeof(data-type) * Array-Size`

Here `sizeof` is a library function, `data-type` is the data type of the array. `Array-Size` specifies the total number of elements the array can take. The result returned by the expression is the total number of bytes the array takes. Another way to calculate the total bytes of an array is simpler; it uses the following expression:

`sizeof(Array-Name)`

Here `Array-Name` is the name of the array. Consider the below program to know how to calculate the memory space taken by an array.

/*Calculating the size of an array.*/

```
#include <stdio.h>
main()
{
    int total_byte;
    int list_int[10];
    total_byte = sizeof (int) * 10;
    printf( "The size of int is %d-byte long.\n", sizeof (int));
    printf( "The array of 10 ints has total %d bytes.\n",
total_byte);
}
```

OUTPUT

The size of int is 2-byte long.
The array of 10 int has total 20 bytes

1.7 Multidimensional Arrays

So far, all the arrays dealt with have been one-dimensional arrays, in which the dimension sizes are placed within a pair of brackets (`[` and `]`). In addition to one-dimensional arrays, the C language also supports multidimensional arrays. Arrays can be declared with as many dimensions as compiler allows. The general form of declaring a `N`-dimensional array is

`data-type Array-Name[Array-Size1][Array-Size2] . . . [Array-SizeN];`

where `N` can be any positive integer.

Because the two-dimensional array, which is widely used and is the simplest form of the multidimensional array, focus is on two-dimensional arrays are focused in this section. However concept of two dimensional array can be applied to arrays of more than two dimensions.

1.7.1 Declaration of Two-dimensional Array

Two-dimensional array is also called as array of arrays a table, a grid or a matrix. Many applications it is required to manipulate the data in table format or in matrix format which contains rows and columns. In order to create such two dimensional arrays in C, following syntax is followed.

```
datatype arrayname[rows][columns];
```

For example, the following statement declares a two-dimensional integer array:

```
int x[2][3];
Array type: int
Array name: x
Array indices: i,j
Standard identification: x[i][j]
Number of entries: n x m
```

Where n is row size and m is column size and x[i][j] indicates any particular element.

Consider the array `int x[2][3];`

Here there are two pairs of brackets that represent two dimensions with a size of 2 and 3 integer elements, respectively. The first dimension 2 is number of rows and second dimension 3 is number of columns. This will create total 6 storage locations for two dimensional arrays as shown below.

| | | |
|---------|---------|---------|
| x[0][0] | x[0][1] | x[0][2] |
| x[1][0] | x[1][1] | x[1][2] |

Arrays can be declared with the other variables in the program, such as

```
float sale[9] [3];
double rate[5] [6];
char name[2] [4];
```

1.7.2 Two-Dimensional Array Initialization

A two dimensional array *data* can be declared and initialized follows:

```
//declaration
int data [3][4] = {
    {8, 2, 6, 5},    //row 0
    {6, 3, 1, 0},    //row 1
    {8, 7, 9, 6}     //row 2
};
```

Similar to the one-dimensional the two-dimensional arrays are initialized. For example

```
int array[2][3] = {1,2,3,4,5,6};
```

In the above case elements of the first row are initialized to 1,2,3 & second row elements are initialized to 4,5,6.

The initialization can be done row wise also, for the above example it is

```
int array[2][3] = {{1,2,3},{4,5,6}};
```

If the initialization has some missing values then they are automatically initialized to 0.

For example

```
int array[2][3] = {{3,4},{5}}
```

In this case the first two elements of the first row are initialized to 3, 4 while the first element of the second row is initialized to 5 & rest all elements are initialized to 0.

User can store the values in each of these memory locations by referring their respective row and column number as,

```
table[2][3] = 10;
table[1][1] = -52;
```

In one dimensional arrays could be initialized without mentioning the size. For example,

```
int a[ ] = {1,2,3}; is same as int a[3] = {1,2,3};
```

Can this be done for two dimensional arrays also? For example consider a multidimensional array sap declared as given below.

```
int sap[ ][ ] = {1,2,3,4,5,6};
```

This is illegal because the array sap could be of 2 rows and 3 columns, or of 3 rows and 2 columns. To avoid this ambiguity, the column size (second subscript) must be specified explicitly in the declaration. For example,

```
int sap [ ][2] = {1,2,3,4,5,6};    /*this is valid /
```

Array sap has two columns and three rows.

But declaring, `int sap [2][] = {1,2,3};`

will not work !

1.8 Processing a Two Dimensional Array

Arrays are usually processed using a **for** statement. To process all the items in a two-dimensional array, a nested **for** statement (that is, one **for** inside another **for**) has to be used. The first **for** loop will process the row indexes of the array, and the inside **for** loop will process the column indexes of the array. The **length** of a two dimensional array is the number of rows it has.

1.8.1 Entering Data into Two Dimensional Array

1) Data can be entered into two dimensional arrays in the following ways. Number of rows is the conditional expression in the outer for loop and number of columns is the conditional expression in the inner for loop.

```
int arr[5][5];
int z = 0, x, y ;
for(x=0; x<5; x++)
{
    for(y=0; y<5; y++)
    {
        arr[x][y] = z;
        z++;
    }
}
```

This will initialize the array 'arr' as,

| | | | | |
|----|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 | 9 |
| 10 | 11 | 12 | 13 | 14 |
| 15 | 16 | 17 | 18 | 19 |
| 20 | 21 | 22 | 23 | 24 |

```
2) int n,m;
   double x[5][6];
```

```
Printf("Enter the elements for the array matrix ");
```

Outer

```
for (n = 0; n < 5; ++n)
{
    _____ → Row index= number of rows
```

```

for (m = 0; m < 6; ++m) → Column index=number of columns
scanf ( "%lf", &x[n][m]);
}

```

1.8.2 Printing data of Two Dimensional Arrays

To output the Array's Data the same concept of two for loops are used. Number of rows is the conditional expression in the outer for loop and number of columns is the conditional expression in the inner for loop. Instead of scanf, printf statement is used inside the inner for loop. This is shown in the example below.

```

int n,m;
double x[5][6];
for (n = 0; n < 5; ++n
{
    for (m = 0; m < 6; ++m)
    printf ( "%10.2f", &x[n][m]);
    Printf("/n ")
}

```

To understand the processing of two dimensional matrix consider the below programs.

1) /*Write a program to create a matrix and display a matrix. */

```

#include <stdio.h>
#include <conio.h>
void main()
{
    int a[2][3],i,j;
    clrscr();
    printf("\nENTER VALUES FOR THE MATRIX:\n");
    for(i=0;i<2;i++)
    for(j=0;j<3;j++)
    scanf("%d",&a[i][j]);
    printf("\nTHE VALUES OF THE MATRIX ARE:\n");
    for(i=0;i<2;i++)
    {
        for(j=0;j<3;j++)
        printf("%5d",a[i][j]);
        printf("\n");
    }
}

```

Result:

Input:

ENTER VALUES FOR THE MATRIX: 1 2 3 4 5 6

Output:

THE VALUES OF THE MATRIX ARE:

1 2 3

4 5 6

2) /*Addition of matrices Program */

```
#include<stdio.h>
main()
{
    int x[][] = { {8,5,6},
                  {1,2,1},
                  {0,8,7}
                };
    int y[][] = { {4,3,2},
                  {3,6,4},
                  {0,0,0}
                };
    int i,j;
    printf("First matrix: ");
    for(i=0;i<3;i++)
    {
        for(j=0;j<3;j++)
            printf("%d", x[i][j]);
        printf("\n");
    }
    printf("Second matrix: ");
    for(i=0;i<3;i++)
    {
        for(int j=0;j<3;j++)
            printf("%d", y[i][j]);
        printf("\n");
    }
    printf("Addition: ");
    for(i=0;i<3;i++)
    {
        for(int j=0;j<3;j++)
            printf("%d", x[i][j]+y[i][j]);
        printf("\n");
    }
}
```

Result:

Input:

First matrix:

```
8 5 6
1 2 1
0 8 7
```

Second matrix:

```
4 3 2
3 6 4
0 0 0
```

Output:

Addition:

```
12 8 8
4 8 5
0 8 7
```

1.9 Passing Arrays as Arguments

When an array is passed to a function as an argument, the array's address is actually passed, rather than the values of the elements in the array. If the function modifies the array values, the array will be modified, since there is only one copy of the array in memory. To pass one dimensional array to the function following rules are considered.

- 1) Function must be called by using only the name of array. It need not include size of array(i.e square brackets[]) or any arguments. For example,

```
int list[] = {0, 4, 5, -8, 17, 301}; /*This defines the array list */
Let biggest be the called function. To pass the array list and an integer
variable to the called function, it can be written as
```

```
biggest(list, n)
```

To pass array alone to the called function, it can be written as
biggest (list)

- 2) The function header must have the data type of the array. Size of array need not be specified. However empty square brackets must be used, to specify the parameter as an array. Function definition can have formal parameters.

For example:

The function header for the called function biggest can be written as

```
int biggest(int array[], int size)
```

The function `biggest` is defined to have two arguments one array name and other an integer value. Here `int array[]` is a formal parameter which is same as actual parameter `int list[]`. `int size` is also a formal parameter.

- 3) Function prototype(if any), must specify the argument is an array. For example,

```
int biggest(int array[], int size);
```

[Remember that the difference between function header and function prototype is only semicolon(;)]

To understand the passing of one dimensional array to a function consider the below two programs.

- 1) /*This program declares one dimensional array and passes it as parameter to a function. The function takes an index and an array, and returns the array element specified by the index: */

```
#include <stdio.h>
int retrieve_element(int index, int array1[]);
main()
{
    int value;
    int a[] = {0, 4, 12, 17, 34};
    value= retrieve_element(3, a);

    Printf("The value of array element is %d ", value);
}
int retrieve_element(int index, int array1 [])
{
    return array1[index]; /* This will return the value of the
element in the array at the position indicated by the value
of index. */
}
```

Output:

The value of array element is 12

Analysis:

(i) Consider the function prototype.

```
int retrieve_element(int index, int array1[]);
```

This denotes that called function name is retrieve-element, its returns integer value to caller function and it has two formal parameters, one is integer variable index and other is int data type array.

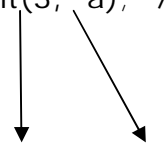
(ii) Function main declares integer array int a[] and integer variables index and value. It calls the function retrieve-element by passing two actual parameters ,first parameter an integer value 3 and second is an array a. The returned result is stored in variable value.

(iii) Function header in the called function has two formal arguments, int index and int array1[] which are same as actual arguments in function 3 and a respectively. In other words this can be described as below.

```
main()
{
    int value;
    int a[] = {0, 4, 12, 17, 34};

    value= retrieve_element(3, a); /* actual parameters where 3=int
value,
    .....
}

int retrieve_element(int index, int array1 []) /* Formal parameters where
                                           3=index, int array1=int
a[] */
{
    .....
}
```



(iv) Consider the called function body

```
return array1[index];
```

Here the index is 3. This statement is equivalent to return array1[3]; This will return the value of the element in the array at the position indicated by the value of index which is twelve in this case.

2)/*This program reads and prints an array elements by passing the array as parameter to function*/

```
#include<stdio.h>
void read(int take[],int in);
void dis(int give[],int out);
void main()
{
    int a[5];
```

```

printf("Enter five elements of first list \n");
read(a,5);
printf("The elements of first list are \n");
dis(a,5);
}
void read(int take[],int in);
{
    int j;
    for(j=0;j<in;j++)
        scanf("%d",&take[j]);
}
void dis(int give[],int out)
{
    int j;
    for(j=0;j<out;j++)
        printf("%d ",give[j]);
    printf("\n");
}

```

Result:

Input:

Enter five elements of first list: 1 2 3 4 5

Output:

The elements of first list are

1
2
3
4
5

1.10 Questions

- 2) What is array? How it is declared?
- 3) Explain the need for array variables.
- 4) Explain two types of initialization of array with an example.
- 5) Explain functioning of an array with example.
- 6) How to define and declare an array?
- 7) Explain how to declare and initialize two dimensional array?
- 8) How an array can be passed as argument?
- 9) Explain the processing of two dimensional array?

1.11 Programing Exercises

- 1) Write a program to create an array. Print the values and addresses of each elements of the array.

- 2) Write a program to create an array of student's ages. Print the average age.
- 3) Write a program to create an array. Print the highest and lowest number in the array.
- 4) Write a program to insert a given number in the array in a given position.
- 5) Write a program to create a matrix and display a matrix.
- 6) Write a program to create two matrixes. Add the values of the two matrixes and store it in another matrix. Display the new matrix.

Module 4

UNIT 2: Strings

2.1 Introduction

The string is sequence of characters that is treated as a single data item. No intrinsic string data type exists in c. The character array ended with a null character \0 is implemented as string. A series of characters enclosed in double quotes (" ") is called a string constant, a string or string literal. The C compiler can automatically add a null character (\0) at the end of a string constant to indicate the end of the string. For example, the character string "hello" is considered a string constant.

2.2 Declaring a string

C does not support string as the data type. However it allows us to represent strings as character arrays. It can be declared as –

```
char string_name[size];
```

Here char is character data type. This is must for specifying the data type as string. string_name is the valid variable name given to the string and 'size' determines the number of characters in the string. For example:

```
char star[10];
char book[13];
char phone[ ];
```

Analysis:

Here 'star' is the character array or a string that can store up to 10 elements of type char. It can be represented as:


```
char star[10];
```

```
[0] [1] [2] [3] [4] [5] [6] [7] [8] [9]
```

However star can also store shorter sequences than 10 characters. Similarly 'book' can store 13 characters. It can be represented as

```
[0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13]
```

```
char phone[ ];
```

Here phone array does not have size specified. This initializes an up-sized character array, phone, with a string constant. Later when the compiler sees the statement, it will figure out the total memory space needed to hold the string constant plus an extra null character added by the compiler itself and allocates the memory space.

2.3 Initializing a String

As other type of arrays, strings can also be initialized at compile time and at run time. The process of initializing the strings at compile time is to write the string literal within double quotes. Compile time initialization is called as static initialization. Compile time initialization can be done in three ways. They are given below.

Type 1. `char name[9] = "Sri Rama";`

Type 2. `char stream[7] = {`H', `e', `l', `l', `o', `!', `\0'};`

Type 3. `char str[] = "I like C.";`

2.3.1 Type 1

Consider a character array name that is initialized with a string constant "Mr JOHN". This is written as
`char name[9] = "Mr JOHN";`

This string is stored in the memory as given below.

| | | | | | | |
|---|---|--|---|---|---|---|
| M | R | | J | O | H | N |
|---|---|--|---|---|---|---|

Here null (\0) is appended by the compiler at the end string to denote that it is end of given string. The compiler can automatically append a null character (\0) to the end of the array and treat the character array as a character string. Note that the size of the array is specified to hold up to nine elements, although the string constant has only eight characters enclosed in double quotes. The extra space is reserved for the null character that the compiler will add later.

```
char str1[4] = "text"; /* illegal declaration*/
```

Here the array `str1` has size 4. This it cannot hold a string constant plus an extra null character and above declaration is considered illegal.

Note that many C compilers will not issue a warning or an error message on this incorrect declaration. The runtime errors that could eventually arise as a result could be very difficult to debug.

2.3.2 Type 2

A character array can also be declared and initialized like this:

```
char stream[6] = {'H', 'e', 'l', 'l', 'o', '!'}; /* char array*/
```

This same array can be declared as string when `null(\0)` is appended at the end by user and array size is made sufficient to hold all characters. This is shown below.

```
char stream[7] = {'H', 'e', 'l', 'l', 'o', '!', '\0'}; /*string */
```

You can also initialize the above a character array *stream* can be initialized with a string constant "Hello! ". For example,

```
char str[7] = "Hello!";
```

The compiler can automatically append a null character (`\0`) to the end of the array, and treat the character array as a character string.

2.3.3 Type 3

The third type of initialization is not specifying the array size. For example,

```
char str[] = "I like C.";
```

This statement initializes an character array, `str`, with a string constant. " I like c. "Later when the compiler sees the statement, it will figure out the total memory space needed to hold the string constant plus an extra null character added by the compiler itself.

2.4 Initialization after Declaration

```
char s[10] = " Web Design";
```

is not the same as

```
char s[10];
```

```
s = "Web Design"; /*Illegal declaration. */
```

This generates a compiler error. The assignment operator cannot be used with a string already declared.

2.5 String Constants versus Character Constants

A string constant is a series of characters enclosed in double quotes (" "). On the other hand, a character constant is a character enclosed in single quotes (` `). When a character variable `ch` and a character array `str` are initialized with the same character, `x`, such as the following,

```
char ch = `x`;
char str[] = "x";
```

One byte is reserved for the character variable `ch`, and two bytes are allocated for the character array `str`. The reason that an extra byte is needed for `str` is that the compiler has to append a null character to the array.

2.6 Input data to string

This is also called as run time initialization. The integers, character and float data types are read by using functions such as `scanf()`, `gets()`, etc. Similarly to read a string two functions are used. They are

1. `scanf()`
2. `gets()`

2.6.1 Reading Strings Using `scanf ()`

To read strings using `scanf()` format specifier `%s` is used with the function `scanf()`. For example

```
char name[10];
```

To input data in to char array 'name', `scanf` function and `%s` is used as given below.

```
scanf("%s",name);
```

Note that the `%s` format specifier doesn't require the ampersand (&) symbol before the variable **name**. The major limitation of `scanf()` is that it reads until occurrence of first separator a white space characters such as space character, tab or new line and store it into given variable. For example if the input literal is "How are you?". Then the `scanf` statement will read only the word 'How' into the variable **name**.

To overcome this limitation, `gets()` and `getchar()` functions are used.

2.6.2 Reading Strings using `gets()`

The `gets()` function reads the string from the keyboard including spaces until the enter key is not pressed. The `getchar()` function is used to get a single character. The `gets()` and `getchar()` functions are defined in `stdio.h` library file. For example

```
char line[100];
printf("Enter a line:\n");
gets(line);
puts("Entered input is :\n");
puts(line);
```

The `gets()` function keeps reading characters from the standard input stream until a new line character or end-of-file is encountered. Instead of saving the new line character, the `gets()` function appends a null character to the array that is referenced by the argument to the `gets()` function.

2.7 Print strings

To output data of the given string two functions are used. They are

- 1) `printf()`
- 2) `puts()`

2.7.1 Printing strings using `printf()`

The `printf` statement along with format specifier `%s` is used to print strings on the screen. The format `%s` can be used to display an array of characters that is terminated by the null character. For example,

```
printf("%s", name);
```

can be used to display the entire contents of the array `name`.

`printf` expects to receive a string as an additional parameter when it sees `%s` in the format string. This additional parameter

- Can be from a character array.
- Can be another literal string.
- Can be from a character pointer (more on this Pointer chapter).

`printf` knows how much to print out because of the NULL character at the end of all strings. When `printf()` encounters a null `\0`, it stops printing. The string variable can be printed out with precision using `printf()` statement. For example

```
printf("%7.3s", name)
```

This specifies that only the first 3 characters have to be printed in a total field width of 7 characters & right justified in the allocated width by default. We can include a minus sign to make it left justified (%-7.3). For example consider the following program.

- `printf("|%5s|","Have a Happy Day");`
 - | Have a Happy Day|
- `printf("|%-5s|","Have a Happy Day");`
 - |Have a Happy Day |
- `printf("|%25.12s|","Have a Happy Day");`
 - | Have a Happy|
- `printf("|%.12s|","Have a Happy Day");`
 - |Have a Happy|

2.7.2 Printing strings using puts()

The puts function is a much simpler output function than printf for string printing. Prototype of puts is defined in stdio.h For example:

```
char sentence[] = "This is strings in C \n";
puts(sentence);
```

This Prints out:

This is strings in C

Consider a simple program to understand string input and output functions.

```
#include <stdio.h>
main()
{
char str[80] ; Char str2[45];
printf("Enter a string: ") ;
gets(str1) ;
printf("Enter another string: ") ;
scanf("%s",str2); /*Reads upto first white space character*/
Printf("output of first string:\n");
printf("%s" , str1\n);
Printf("output of second string:\n");
puts("str2 ")
}
```

Result:

Input:

Enter a string: This is c language.

Enter another string: This is strings

Output:

Output of first string: This is c language

Output of second string: This

2.8 Built-in String Functions

C does not provide any operators for string. Because of this C has built in string functions in its library. C library supports a large number of string handling functions that can be used to perform many of the string manipulations. Some of the string functions are given below. To do all the operations described here it is essential to include string.h library header file in the program.

Some of the String Functions in string.h

| Function name | Description |
|----------------------|--|
| strcpy(str1,str2) | copies str2 to str1 including the null (\0). |
| strcat(str1,str2) | Appends str2 to end of string str1 |
| strlen(string) | Returns the length of string. Does not include the null(\0) |
| strcmp(str1,str2) | Compares str1 with str2 returns integer result.If str1<str2 returns negative integer.Returns zero when str1==str2,and Returns a positive integer when str1>str2. |
| Strncpy(str1,str2,n) | It copies at most n characters of str2 to str1.If str2 has fewer than n characters, it pads str1 with null('\0') characters. |
| Strchr(string,char) | Locates the position of the first occurrence of char within string and returns the address of the character if it finds. and null if not.For ex-(" Hello", 'l') |
| strlwr() | converts all characters in a string from uppercase to lowercase. |
| strupr() | converts all characters in a string from lower case to uppercase. |

strcpy() function:

C does not allow you to assign the characters to a string directly.For example as in the statement name="Java"; Instead use the strcpy() function. The strcpy() function works almost like a string-assignment operator. The syntax of the function is

```
strcpy(string1,string2);
```

Strcpy() function assigns the contents of string2 to string1.The string2 may be a character array variable or a string constant.

```
strcpy(Name,"Java");
```

```
strcpy(city1, city2);
```

This will assign the contents of the string variable city2 to the string variable city1. The size of the array city1 should be large enough to receive the contents of city2, if not compiler will lodge an error.

strcat() function:

The process of appending the characters of one string to the end of other string. Is called concatenation. The strcat() function joins two strings together. Sstrcat(string1, string2)

Here string1 & string2 are character arrays. When the function strcat is executed string2 is appended to string1. The string at string2 remains unchanged.

Example

```
strcpy(string1, "Web");
strcpy(string2, "Mining");
Printf("%s", strcat(string1, string2));
```

From the above program segment the value of string1 becomes webmining. The string at str2 remains unchanged as mining. The variation in this function is strn(str1, str2, n). This concatenates the n integer letters to str1. For example,

```
strncat(stri, name, 4);
```

This concatenates first 4 letters of string name to string stri.

strlen() function:

This function counts and returns the number of characters in a string. The length does not include a null character. The syntax is n= strlen(string); Where n is integer variable, which receives the value of length of the string.

Example:

```
int len;
```

```
len= strlen("Coral Draw");
```

Here the integer variable len will store the value 10 which is the length of given sting.

strcmp() function

In c the values of two strings cannot be directly compare in a condition like if(string1==string2). The strcmp() function is used for this purpose. This function returns a zero value if two strings are equal, or a non zero

number if the strings are not the same. This returns a negative if the string1 is alphabetically less than the second and a positive number if the string is greater than the second.

The syntax of strcmp() is given below:

Strcmp(string1,string2)

String1 & string2 may be string variables or string constants.

Example:

strcmp("Book","Book") will return zero because 2 strings are equal.

strcmp("their","there") will return a 9 which is the numeric difference between ASCII 'i' and ASCII 'r'.

strcmp("The", "the") will return 32 which is the numeric difference between ASCII "T" & ASCII "t".

strcmapi() function

This function is same as strcmp() which compares 2 strings but not case sensitive. For example

strcmapi("THE", "the"); will return 0.

strlwr () function:

This function converts all characters in a string from uppercase to lowercase. The syntax is

strlwr(string);

For example:

strlwr("APPLICATION") converts the string to application

strupr() function:

This function converts all characters in a string from lower case to uppercase. The syntax is

strupr(string);

For example strupr("application") will convert the string to APPLICATION.

2.9 C Character Functions

Apart from string functions C has character functions. These functions are defined in ctype.h library file. Some of the character functions are given below.

| <u>Function</u> | <u>Return true if</u> |
|-----------------|-----------------------|
| int isalpha(c); | c is a letter. |


```

int isupper(c);    c is an upper case letter.
int islower(c);    c is a lower case letter.
int isdigit(c);    c is a digit [0-9].
int isxdigit(c);   c is a hexadecimal digit [0-9A-Fa-f].
int isalnum(c);    c is an alphanumeric character (c is a letter or a digit);
int isspace(c);    c is a SPACE, TAB, RETURN, NEWLINE, FORMFEED,
                  or vertical tab character.
int ispunct(c);    c is a punctuation character (neither control nor
                  alphanumeric).
int isprint(c);    c is a printing character.
int iscntrl(c);    c is a delete character or ordinary control character.
int isascii(c);    c is an ASCII character, code less than 0200.

int  toupper(int convert character c to upper case (leave it alone if not
c);                      lower)
int  tolower(int convert character c to lower case (leave it alone if not
c);                      upper)

```

Programs

1. Write a program to find out the length of a given string without using the library function `strlen()`.

```

#include <stdio.h>
void main()
{
    char str[50];
    char nul={'\0'}
    int len;
    printf("\nENTER A STRING: ");
    gets(str);
    for(len=0; str[len]!='\0'; len++);

    printf("\n THE LENGTH OF THE STRING IS %d", len);
}

```

Result:

Input:

ENTER A STRING: This is string in C

Output:

THE LENGTH OF THE STRING IS 19.

2) Write a program to find out the length of a given string using the library function `strlen()`.

/*The "strlen()" function gives the length of a string, not including the NULL character at the end.*/

```
#include <stdio.h>
#include <string.h>
void main()
{
    char name[30] = "This is string in C";
    int len;
    len= strlen( name );
    printf( "Length of string <%s> is %d.\n", name ,len);
}
```

Output:

Length of string < This is string in C > is19.

3) Write a program to print the reverse of a given string.

```
#include <stdio.h>
#include <string.h>
void main()
{
    char ch[100];
    int i,len;
    printf("\n Enter a string: ");
    gets(ch);
    len= strlen(ch);
    printf("\n The string in the reverse order: ");
    for(i=len-1; i>=0; i--)
        printf("%c", ch[i]);
    getch();
}
```

Result:

Input

Enter a string: strings are interesting.

Output

The string in the reverse order: .gnitseretni era sgnirts

4)Write a program to check if a given string is palindrome or not.

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

```

void main()
{
    char a[100];
    int i, len, flag=0;
    printf("\n Enter a string: ");
    gets(a);
    len=strlen(a);
    for(i=0; i<len; i++)
    {
        if(a[i]==a[len-i-1])
            flag=flag+1;
    }
    if(flag==len)
        printf("\n The string is palindrom");
    else
        printf("\n The string is not palindrom");
}

```

RESULT:

Input

Enter a string: madam

Output

The string is palindrome

5) Write a program to count the number of vowels in a given string.

```

#include <stdio.h>
#include <string.h>
void main()
{
    char a[100];
    int len, vow=0;
    printf("\n ENTER A STRING: ");
    gets(a);
    len=strlen(a);
    for(i=0; i<len; i++)
    {
        if(a[i]=='a' || a[i]=='A' || a[i]=='e' || a[i]=='E' || a[i]=='i' ||
           a[i]=='I' || a[i]=='o' || a[i]=='O' || a[i]=='u' || a[i]=='U')
            vow=vow+1;
    }
    printf("\n There are %d vowels in the string", vow);
}

```

RESULT:

Input

Enter a string: Evening

Output

There are 3 vowels in the string

6) Write a program to count the number of words in a given string. Two words are separated by one or more blank spaces.

```
#include <stdio.h>
#include <string.h>
void main()
{
    char a[100];
    int len, word, i = 0;
    printf("\n Enter a string: ");
    gets(a);
    len = strlen(a);
    for(i = 0; i < len; i++)
    {
        if( a[i] != ' ' && a[i+1] == ' ')
            word = word + 1;
    }
    printf("\n There are %d words in the string", word);
}
```

RESULT:

Input

Enter a string: Strings are very interesting.

Output

There are 4 words in the string

2.10 Questions

- 1) What is a string? How strings are different from character strings. Give different types of initialization of the strings.
- 2) Explain different functions used to write and read the strings.
- 3) Explain any five string handling functions.
- 4) Differentiate between scanf() and gets() pertaining to reading the strings.

2.11 Programing Exercise

1. Write a program to copy contents of one array into another array.

Accept size of both arrays as 7.

2) Read a string from keyboard and inverse the case of it. That is, convert lower case letters to upper-case and vice versa.

3) Input any two strings from user and find whether these are equal or not.

4) Read a string from user and analyze it. That is, count total number of alphabets, digits, special symbols and display their counts.

5) Write a program to accept a string and display a list of ASCII codes which represent accepted string.

Module 5

Unit 1 : POINTERS

1.1 Introduction

Pointers are a fundamental part of C. A pointer is a derived data type in 'C'. It is built from any one of the primary data type available in 'C' programming language. Pointers contain memory addresses as their values. C uses pointers because of following advantages namely,

- It is the only way to express some computations.
- It produces compact and efficient code.
- It provides a very powerful tool to solve complex problems.
- It reduces the complexity and length of the program.

C uses pointers explicitly with:

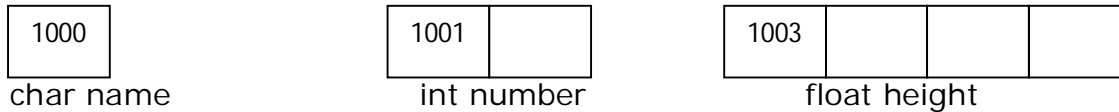
- Arrays
- Structures
- Functions

1.2 How variables are stored in memory?

Consider the following variables declared in a program:

```
char name;
int number;
float height;
```

When a variable is defined the compiler allocates a real memory address for the variable. For character one byte, float four bytes and int two bytes are allocated generally. The following diagram illustrates how the above declared variables might be arranged in memory along with their addresses



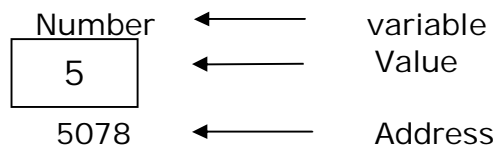
When the variables are initialized, the value of the variable gets stored in that respective memory location.

1.3 What is a Pointer?

A pointer is a special variable that holds the address of the memory location of another variable. Another variable could be anything. It could be a float, int, char, double, etc. Consider the following statement:

```
int number =5;
```

For integer variable quantity let the address of the variable be 5078. This can be diagrammatically presented as given below.



During the execution of program computer always associates the integer variable number with address 5078. The value 5 can be accessed by using either the variable number or the address of it which is 5078. Since memory addresses are simply numbers they can be assigned to some variables which can be stored in memory like any other variables. Such variables that hold memory addresses are called as 'pointer variables' the pointer variable is nothing but a variable that contains an address of another variable in memory. When a pointer contains address of integer variable, such pointer is called *integer pointer*. Similarly float, char pointers contain the address of char and float data type.

1.4 Pointer Declaration

In C every variable must be declared before it is used. Since pointer is a special variable, its declaration is also different. An operator called dereference operator or indirection operator is used to declare a Pointer. It is represented by *(asterisk). The '*' symbol appears in C language in four different situations with four different meanings. Two of these are well known.

- (i) Comments /*...*/
- (ii) Arithmetic operator for multiplication as in a*b
- (iii) Declaring pointer variable using *(dereference operator)

Usage of deference operator in comment can be easily recognized. Multiplication needs two operands; this can also be easily recognized. When it appears in a declaration as `int *p`, read it is read as `p` is a pointer to an integer. The syntax of pointer declaration is

`data-type *variable-name;`

Here data type can be `int`, `float`, `char` etc and the dereference operator before the `int` variable `variable-name` tells that it is a pointer variable. Hence the pointer variable `variable-name` holds the address of another variable of the same data type. For example,

A pointer is declared by assigning an asterisk (*) in front of the variable name in the declaration statement.

```
int x;      /* define x */
int *ptr;   /* define a pointer to x */
int *p;     → p is a pointer to an integer.
            similarly pointer to char and float can be declared
char *q;    → q is a pointer to a char.
float *fp;  → fp is a pointer to float.
```

The "dereferencing operator" the asterisk can also be is used as follows:

```
*ptr = 7;
```

This will copy 7 to the address pointed to by **ptr**. Thus if **ptr** "points to" (contains the address of) **x**, the above statement will set the value of **x** to 7. That is, when the '*' is used this way, it is referring to the value of that which **ptr** is pointing to, not the value of the pointer itself.

1.4.1 Pointer Declaration Styles

The dereference operator (*) can appear anywhere between the data type name and the pointer variable name as shown below.,

```
int * p; /*style 1 between the data type and variable*/
int *p; /*style 2 Close to variable*/
int* p; /*style 3 close to data type.*/
```

1.4.2 Multiple Declarations

Multiple pointers of same data type can be declared as given below.

```
int *p, x, *q;
```

Here **p** and **q** are pointer variables and **x** is integer variable.

1.5 Pointer Initialization

Consider the declaration

```
int i, j, *p;
```

- It declares a pointer variable *p* that can point to an integer. Just this kind of declaration, doesn't specify where a integer pointer *p* has to point at particular. In other words it does not say anything about which particular integer variable address it has to hold, either of *i* or of *j*, in this case. To explicitly specify where variable a pointer has to point to, a unary operator called **reference or address operator (&)** is used. The reference operator (&) cannot be applied to expressions, constants, or register variables. To initialize the pointer variable reference operator (&) is used with pointer variable. To understand different types of initialization consider the following examples,

```
(i)   int i, j, *p;
        p = &i;
```

This causes *p* pointer variable to point at *i* the integer data type. Here

- * is the *indirection* operator
- & is the *address* operator

1.5.1 The Address of Variables

All the variables that are declared in programs are allocated addresses in the memory. This address can be printed out using the & operator. (which has already been used in `scanf()`). For example, consider the below program.

```
#include <stdio.h>
main()
{
    int x;
    x=1000;
    printf(" Value of x=%d\n",x);
    printf("Address of variable x = %d\n", &x);
}
OUTPUT:
```

```
Value of x = 1000
Address of variable x = 278614
```

Analysis:

In this case, the value will be stored in 4 locations, not one, since integer requires 4 bytes to store. However 1000 is 'sliced' into 4 pieces,. Here, what is the address of *x*? Actually it is 278614, 278615, 278616 and

278617. However in C we consider the address of C as 278614, the address of first of the four locations. This is an important thing to remember and crucial to the understanding of pointers

```
(ii)  int x;
      int *ptr;
      ptr=&x; /*Address of x is assigned to pointer variable*/
      *ptr=&x; /*The value of x is assigned to ptr */
```

1.5.2 Null Pointers

A pointer is said to be a null pointer when its value is 0. Remember, a null pointer can never point to valid data. To set a null pointer, simply assign 0 to the pointer variable. For example:

```
char *pc;
int *pt;
pt=pc=0;
```

Here pt and pc pointers become null pointers after the integer value of 0 is assigned to them. Later in the program any required value can be assigned to null pointers. This analogous to initializing the variables to zero value in the program, such as

```
int x=0;
X=5;
```

1.5.3 Understanding pointers

Pointers has the following two important aspects. They are given below.

&x = address of variable x.

*p = content of address given by p.

Pointers concept is well understood by following example problems.

(i) Write a program to display the contents of a pointer.

```
#include <stdio.h>
int main()
{
    int x = 12;
    int *ptr;
    ptr = &x;

    printf("Address of x: 0x%p\n", ptr);
    printf("Address of x: 0x%x\n", &x);
    printf("Address of ptr: 0x%x\n", &ptr);
    printf("Value of x: %d\n", *ptr);
```

```
    return 0;
}
```

OUTPUT:

Address of x: 0x0065FBFA
 Address of x: 0x65fbfa
 Address of ptr: 0x65fbfa
 Value of x: 12

Note: Format specifiers %x,%X and %p are used to get the address of the variable in c. ox in the address refers that the format of address is in hexadecimal format.

%x -> Hexadecimal value represented with lowercase characters (unsigned integer type)

%X -> Hexadecimal value represented with uppercase characters (unsigned integer type)

%p -> Displays a memory address (pointer type) compatible to the computer memory.

2) Write a simple program to understand the usage deference and reference operators in pointers

```
#include <stdio.h>
int main()
{
    int *ptr, q;

    q = 19; /* assign 19 to q */

    ptr = &q; /* assign ptr the address of q */
    printf(" Vlaue of q=%d\n",q);
    Printf("Contents of ptr=%d\n ",*ptr);
    printf("Address of q stored in ptr=%d ",ptr);
    return 0;
}
```

OUTPUT

value of q=19
 Contents of ptr=19
 address of q stored in ptr=782AB2

3) Write a program to assign a character variable to the pointer and to display the contents of the pointer.

```
#include <stdio.h>
void main()
```

```

{
    char x, y;
    char *p;
    x='t';
    p=&x;
    y=*p;
    printf("value of x=%c\n", x);
    printf("Pointer value in y=%d\n", y);
    printf("Pointer value in *p=%d\n", *p);
}

```

Output

Value of x=t

Pointer value in y=t

Pointer value in *p=t

4) Write a program to assign the pointer variable to another pointer and display the contents of both pointer variables.

```

#include<stdio.h>
main()
{
    int x;
    int *ptr1,*ptr2;
    x=5;
    ptr1=&x;
    ptr2=ptr1;

    printf("value of x=%d\n", x);
    printf("Contents of ptr1=%d\n",*ptr1);
    printf("Contents of ptr2=%d\n", *ptr2);
}

```

Output

Value of x=5

Contents of ptr1=5

Contents of ptr2=5

1.6 Pointer Expressions

To understand the working of pointers consider the below expressions.

```

int x,y;
int *ptr1,*ptr2;

```

1)ptr1=&x;

The memory address of the variable x is assigned to pointer variable ptr1.

2) `y=*ptr1`

pointer variable ptr1 is holding the address. The content of that address is assigned to the variable y, not the memory address.

3) `ptr1=&x;`

`ptr2=ptr1;`

Ptr1 holds the address of the variable due to first declaration. In the second declaration content of ptr1 ie the address of x is transferred to ptr2. Hence both ptr1, ptr2 both will be holding the same address.

4) For better understanding of the pointers consider some of the invalid declarations given below.

(i) `int x;`

`int ptr1;`

`ptr1=&x;`

Error: pointer declaration must have the prefix of dereference (*) operator.

(ii) `float p;`

`float*var`

`var=y;`

Error: While assigning variable to the pointer variable the address operator (&) must be used along with the variable.

(iii) `int x;`

`char *name;`

`name=&x;`

error: Mixing of data type is not allowed.

1.7 Pointer Arithmetic

Pointer is a variable .Some arithmetic operations can be performed with pointers. C language supports four arithmetic operators which can be performed with pointers .They are

addition +

subtraction -

Pointer increment

Pointer decrement

1.7.1 Pointer increment and decrement

Integer, float, char, double data type pointers can be incremented and decremented. For all these data types both prefix and post fix increment

or decrement is allowed. Integer pointers are incremented or decremented in the multiples of two. Similarly character by one, float by four and double pointers by eight etc.

Let `int*p;`

```
P++      /*valid*/
++p      /*valid*/
p--      /*valid*/
--p      /*valid*/
```

This is illustrated in the below program.

```
#include <stdio.h>
main()
{
    int *p1,p;
    float *f1,f;
    char *c1,c;
    p1=&p;
    f1=&f;
    c1=&c;

    printf("Memory      address      before      increment:\n
    int=%p\n,float=%p\n, char=%p\n",p1,f1,c1);
    p1++;
    f1++;
    c1++;
    printf("Memory address after increment:\n int=%p\n, float=%p\n,
    char=%p\n",p1,f1,c1);
}
```

Output:

Memory address before increment:

```
int=0045AF19      /*int Occupies two bytes*/
float=0045AF2B      /*float occupies four bytes*/
char=          0045AF3E      /*character occupies one byte*/
```

Memory address after increment:

```
int=0045AF1B
float=0045AF2F
char=          0045AF3F
```

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

```

int x;
int *p;
x=1000;
p=&x;
printf("Pointer value = %d\n",p);
printf("Pointer plus one =%d\n",p+1);
}

```

An example output can be:

Pointer value = 22F455

Pointer plus one = 22F439

Analysis:

223455+1=223439! That is pointer magic! How does C justify that? Well, 223455 is not just a number, C knows it is the address of an integer that takes 4 locations. So 223455, 223456, 223457 and 223458 are all together held by the integer. So C interprets +1 as next free location and gives the answer 223459. Remember that it need not be continuation of memory location, when a pointer is incremented.

1.7.2 Pointer Addition and Subtraction

Other than addition and subtraction, no other operations are allowed on integer pointers. Addition and subtraction with float or double data type pointers are not allowed. Pointers cannot be added or subtracted from each other. For example,

```

int*p1,*p2;
p1=p1+p2; /* invalid */
p1=p1-p2; /* invalid */

```

A constant can be added or subtracted from integer pointer variable. For example,

```

int*ptr;
ptr=ptr+9;

```

To understand the pointer additions consider the following program.

/*pointer arithmetic*/

```

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

```

```

int*ptr1,*ptr2;
x=10;
ptr1=&x;
ptr2=ptr1+6;
printf(" Value of x=%d\n",x);
printf("Value pointed by ptr1 =%d\n ",*ptr1);
printf(" Address of x pointed by ptr1=%d\n", ptr1);
printf("Content of ptr2=(ptr1+6)is =%u ",ptr2);
printf("Value pointed by ptr1=%d ", *ptr2);
}

```

OUTPUT:

```

Value pointed by ptr1=10
Address of x pointed by ptr1=00785614
Content of ptr2=(ptr1+6)is = 0078561A
Value pointed by ptr2=8A/* let the value at location 0078561A be 8A
*/

```

1.7.3 Pointer Multiplication and Division

Multiplication or division is not allowed with the pointers. For example

```

int*p1,*p2;
p1=p1*p2; /*Invalid*/
p1=p1/p2; /*Invalid*/

```

Pointer cannot be multiplied or divided a pointer by constant. For example:

```

p = p * 4; /* invalid */
p = p / 2; /* invalid */

```

1.8 Pointers and Function

C allows operations of pointers with functions. The typical use of this is passing an argument to called function in the function declaration. Sometimes only with a pointer a complex function can be easily represented and success. The usage of the pointers in a function definition may be classified into two groups.

1. Call by reference
2. Call by value.

1.8.1 Call by Reference

In this case, the address of a variable is passed to a function through a pointer. This means that the value of the variable maybe changed inside the function. Passing the address of a variable to a function is called *passing by reference*. In order to declare a pointer to a function it has to be declared like the prototype of the function except that the name of the function is enclosed between parentheses () and an asterisk (*) is inserted before the name. The function which is called by reference can change the values of the variable used in the calling program. For example,

```
int x, y;
```

To pass by reference the variables x and y to function add,

```
add (&x, &y);      /* address of x and y is passed*/
```

In the called function add () x and y variable values can be obtained by inserting asterisk (*) before the formal parameters name. For example,

```
main()
{
    int x ,y, z;
    x=10;
    y=20;
    add(&x, &y); /*Call by reference*/
    z=x+y;      /*Z=20+40=60*/
}

add(a,b)
int *a, int *b; /*a,b are formal arguments/
{
    *a=*a+*a; /*New values will be copied to actual parameters in
main*/
    *b=*b+*b;
}
```

```
/* example of call by reference*/
```

```
# include< stdio.h >
void main()
{
    int x,y;
    x=20;
    y=30;
    printf("\n Value of a and b before function call a=%d, b= %d", a, b);
    fncn(&x, &y);
    printf("\n Value of a and b after function call a=%d ,b=%d", a, b);
}
fncn(p,q)
```



```

int *p,*q;
{
*p=*p+*p+2;
*q=*q+*q+3;
}

```

Value of a and b before function call a=20, b=40

Value of a and b after function call a=42, b=63

2) int myfunction(int p1, int* p2)

```

{
    p1 = 1 ;
    *p2 = 2;
}
int main()
{
    int x = 5, y = 6;
    int *pY = &y ;
    printf("\nValue of x and y before function call x=%d, y=%d", x, y);
    myfunction(x, pY) ;
    printf("\n Value of a and b after function x=%d, y=%d", x, y);
}

```

Output:

x=5, y=6

x=1, y=2

1.8.2 Call by Value

Call by value explained well in functions chapter. Here only the value is passed but not address. Hence the changes made to formal arguments do not affect the actual parameters in the main function.

1.9 Pointers and Arrays

An array is actually very much like a pointer. We can declare the arrays first element of array can be declared as a1[0] or as int *a1 because a[0] is an address and *a is also an address. Hence the correspondence between array and pointers can be understood in the following section.

1.9.1 Pointers and One Dimensional Array

An array is a collection of items of the same data type. For example, the following are all array declarations:

```
int x[30]; /* an array of integers/
```

```
char name[20]; /* an array of characters*/
```

```
double g1[30]; /* an array of doubles*/
```

Consider the following:

```
int x[5] = {8, 4, 9, 6, 3};
```

Here array x is containing 5 integers. Each of these integers can be referred by means of a subscript to x i.e. using x[0] through x[5].

Alternatively array can be accessed via a pointer as follows:

```
int *ptr; /*declare the pointer*/
```

```
ptr = &x[0]; /* pointer points to first element of the array*/
```

In the similar way other array elements can also be accessed as given below.

```
p = &x[1]; /* pointer points to second element of the array*/
```

```
p = &x[2]; /* pointer points to third element of the array*/
```

```
p = &x[3]; /* pointer points to fourth element of the array*/
```

The other way of assigning the array to pointer is

```
int p[200];
```

```
int*ptr;
```

```
ptr=p;
```

This is exactly same as ptr=p[0];

The following equalities are also valid.

```
Ptr+6=&p[6];
```

```
*ptr==&p[0];
```

```
*(ptr+6)==&value[6];
```

The array subscripting is defined in terms of pointer arithmetic. The expression a[i] is defined to be same as *(a+(i)).

Array and pointers concept can be well understood by the following program.

```
#include<stdio.h>
main()
{
    int a[4]={1,2,3,4};
    int*ptr;
    int i,n,temp;
    n=4;
    Printf(" Contents of array\n")
    for(i=0;i<=n-1;i++)
    temp=a[i];
    printf("a[%d]value =%d \n" i, temp);
}
```

Output:

Contents of array

a[0] value=1

a[1] value=2

a[2] value=3

a[3] value=4

1.9.2 Pointers and Strings

A string is an array of characters ending with the NULL character. A string constant is enclosed in double quotes, e.g. "hello", "I like C". In order to assign a pointer to string, a character data type pointer variable is declared and the address of the first element of a character array is assigned to pointer variable as shown. For example,

```
char a[] = "hello"; /*string*/
char *ptr1; /*Declare a character data type pointer*/
char *ptr1 = &a[0]; /*Assign array to pointer*/
```

To print the string:

```
printf("string is: %s\n",a);
printf("string pointed by pointer ptr1: %s\n",ptr1);
```

OUTPUT:

```
string is: hello
string pointed by pointer ptr1: hello
```

The concept of string is well understood by the following example.

```
#include <conio.h>
void main() {
    clrscr();
    char *array[2];
    array[0]="Hello";
    array[1]="World";
    printf("The Array of String is = %s,%s\n", array[0], array[1]);
    getch();
}
```

Output

```
The Array of String is = Hello, World
```

All the string functions can be used with pointers. Consider a below program to understand usage of strlen() function with pointer.

```
/* Measuring string length */
#include <string.h>
int main()
{
    char str1[] = { 'A', ' ', 's', 't', 'r', 'i', 'n', 'g', '\0' };
    char str2[] = "string constant";
    char *ptr_str = "Assign a string to a pointer.";
```

```

printf("The length of str1 is: %d bytes\n", strlen(str1));
printf("The length of str2 is: %d bytes\n", strlen(str2));
printf("The length of the string assigned to ptr_str is: %d bytes\n",
strlen(ptr_str));
return 0;
}

```

OUTPUT

The length of str1 is: 9 bytes
 The length of str2 is: 15 bytes
 The length of the string assigned to ptr_str is: 29 bytes

1.10 Questions

1. What is a pointer? what are the uses of pointers in C?
2. 1. Write the following pointer declarations.
 - (a) p, a pointer to an integer
 - (b) char p, a pointer to a character
 - (c) fp, a pointer to a float
3. How is a pointer variable different from an ordinary variable?
4. Differentiate between reference and dereference operators?
5. Explain pointer arithmetic.
6. How do you pass a pointer to a function?
7. Explain call by reference with an example.
8. How pointers are used in arrays? Explain with an example.
9. How pointers are used in strings? Explain with an example.

1.11 Programing Exercise

- 1) Predict the output of each of the following program. Size of memory address can be assumed to 6 digits.

```

a) int a;
int *integer_pointer;
a=222;
integer_pointer=&a;
printf("The value of a a %d\n", a);
printf("The address of a %d\n",&a);
printf("The address of integer_pointer %d\n", &integer_pointer);
printf("Star integer_pointer %d\n", *integer_pointer);

```

```

b) for char
char a;
char *char_pointer;
a='b';
char_pointer=&a;

```

```
printf("The value of a %d\n", a);
printf("The address of a %d\n", &a);
printf("The address of char_pointer %d\n", &char_pointer);
printf("Star char_pointer %d\n", *char_pointer);
```

c) for float

```
float a;
float *float_pointer;
a=22.25;
float_pointer=&a;
printf("The value of a %d\n", a);
printf("The address of a %d\n", &a);
printf("The address of float_pointer %d\n", &float_pointer);
printf("Star float_pointer %d\n", *float_pointer);
```

d) int a, b

```
int *ip1, *ip2;
a=5;
b=6;
ip1=&a;
ip2=ip1;
printf("The value of a is %d\n", a);
printf("The value of b is %d\n", b);
printf("The address of a is %d\n",&a);
printf("The address of b is %d \n",&b);
printf("The address of ip1 is %d\n", &ip1);
printf("The address of ip2 is %d\n", &ip2);
printf("The value of ip1 is %d\n",ip1);
printf("The value of ip2 is %d\n", ip2);
```

2) Write a c program to find a given string in the line of text using a pointer

3) Write a program to swap two numbers using pointers.

4) Write a program to find out whether a given string is palindrome or not using pointers.

5) Write a program to reverse a string using pointers.

Module 5

UNIT 2: Preprocessors

2.1 Introduction

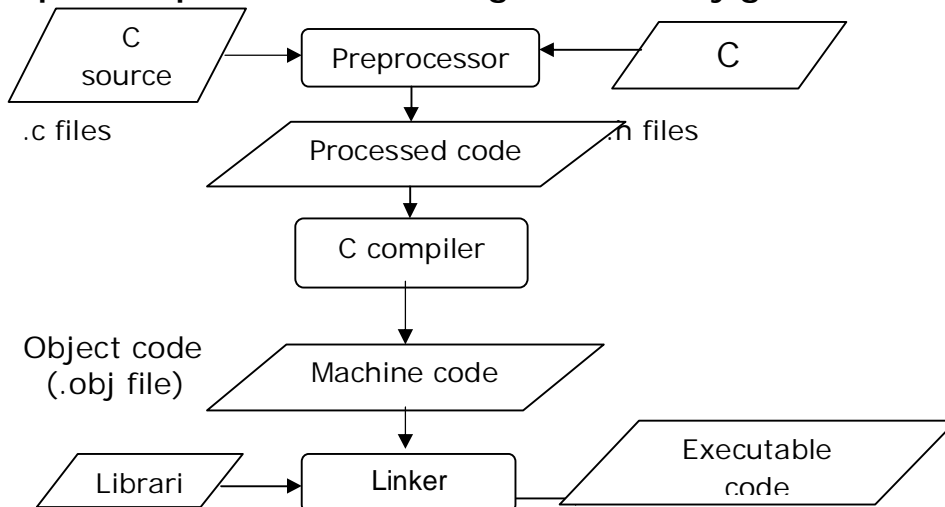
A unique feature of c language is the preprocessor. A program can use the tools provided by preprocessor to make his program easy to read, modify, portable and more efficient.

The C preprocessor is a collection of special statements, called directives that are executed at the beginning of the compilation process. It operates under the control of preprocessor command lines and directives. Preprocessor directives are placed in the source program before the main line before the source code passes through the compiler it is examined by the preprocessor for any preprocessor directives. If there is any appropriate actions are taken then the source program is handed over to the compiler.

Preprocessor directives follow the special syntax rules that are listed below.

- Executed by the pre-processor.
- Occurs before a program is compiled.
- Begin with #.
- Would not end with semicolon.
- Can be placed anywhere in the program.
- Normally placed at the beginning of the program or before any particular function.

The compilation process can be diagrammatically given as below.



Types of Preprocessor Directives

A set of commonly used preprocessor directives are given below.

unconditional directives

| | |
|-----------------------------------|---|
| #include | Inserts a particular header from another file |
| #define | Defines a preprocessor macro |
| #undef | Undefines a preprocessor macro |
| The conditional directives | |
| #ifdef | If this macro is defined |
| #ifndef | If this macro is not defined |
| #if | Test if a compile time condition is true |
| #else | The alternative for #if |
| #elif | #else an #if in one statement |
| #endif | End preprocessor conditional |

2.2 Preprocessor Directives

The preprocessor directives can be divided into three categories. They are

1. Macro substitution directives
2. File inclusion directives
3. Compiler control directives

2.3 Macro Substitution Directives

Macro substitution is a process where an identifier in a program is replaced by a pre defined string composed of one or more tokens. The #define statement is used for this task. It has the following form

#define identifier string

This definition is known as a macro definition. The preprocessor replaces every occurrence of the identifier in the source code by a string. The definition should start with the keyword #define followed by an identifier and a string with at least one blank space between them. Definition is not terminated by a semicolon. The string may be any text and identifier must be a valid c name.

There are three different forms of macro substitution. They are,

1. Simple macro substitution
2. Argument macro substitution
3. Nested macro substitution

Note that in this book only simple macro substitution alone is explained.

2.3.1 Simple Macro Substitution

Simple string replacement is commonly used to define constants. For example:

```
#define PI 3.1415926
#define CAPITAL "BANGALORE"
#define AREA 12.36
```

Writing macro definition in capitals is a convention not a rule. Macro is not ended by semicolon.

Consider the below program to understand the functioning of macro.

```
#include<stdio.h>
#define PI 3.14
#define ERROR_1 "File not found."
#define QUOTE "Hello World!"
main()
{
printf("Area of circle = %f * diameter", PI );
printf("\nError : %s",ERROR_1);
printf("\nQuote : %s\n",QUOTE);
}
```

Preprocessor step is performed before compilation and it will change the actual source code to below code. Compiler will see the program as given below.

```
#include<stdio.h>
main()
{
printf("Area of circle = %f * diameter", 3.14 );
printf("\nerror : %s","File not found.");
printf("\nQuote : %s","Hello World!\n");
system("pause");
}
```

Output

```
Area of circle = 3.140000 * diameter
Error : File not found.
Quote : Hello World!
```

2.3.2 Macro Inside the Quotes

While programming, care should be taken such that macro is not included inside quotes because the macro inside a string does not replaced. For example ,


```
# define P 5
sum= P+ value;
printf("P= %d\n ", P);
```

During preprocessing all the occurrences of P is replaced by 5 except for the P inside the string. In the string " P= %d\n ",P is left unchanged. This is as shown below.

```
sum= 5+ value;
printf("P= %d\n ", 5);
```

2.3.3 Macros With Parameters

The preprocessor permits us to define more complex and more useful form of replacements it takes the following form.

```
# define identifier(f1,f2,f3.....fn) string.
A simple example of a macro with arguments is
# define CUBE (x) (x*x*x)
```

If the following statements appears later in the program,

```
volume=CUBE(side);
```

The preprocessor would expand the statement to
 volume =(side*side*side)

2.3.4 Undefined a Macro

A defined macro can be undefined using the statement #undef<macro name>. For example to undefined a macro, CUBE we can write

```
# undef CUBE
```

This is useful when we want to restrict the definition only to a particular part of the program.

2.4 File Inclusion

The preprocessor directive include< file name> can be used to include any file into user program if the function s or macro definitions are present in the source code. This represented as below.

```
#include< filename >
```

For example,

```

#include<stdio.h>
#include<string.h>
main()
{
    -----
    -----
    printf(" ---");
    x= strcpy(---);

}

```

In the above program printf() is defined in the standard library file stdio.h. and strcpy is defined in string.h. library file. Because of this stdio.h and string.h are used in the file inclusion macro.

2.5 Questions

- 1) what is a macro?
- 2) What is a preprocessor? How it is different from a macro?
- 3) What is a header file in c? What is the purpose of using this in program?
- 4) List out preprocessor directives.
- 5) What are the advantages of macro?

2.6 Programing Exercise

- 1) Write a macro in c to find odd or even number.
- 2) Define a macro name that can multiply two arguments. Write a program to calculate the multiplication of 2 and 3 with the help of the macro. Print out the result of the program.
- 3) Write a symbolic constant or a macro definition for each of the following situations.
 - (a) Define the symbolic constant P I to represent the value 3.141 5927.
 - (b) Define a macro called AREA, which will calculate the area of a circle in terms of its radius.
 - (c) Rewrite the macro described in the preceding problem so that the radius is expressed as an argument.
 - (d) Define a macro called CIRCUMFERENCE, which will calculate the circumference of a circle in terms of its radius. Use the constant PI.

Module 6

Unit 1: STRUCTURES

1.1 Introduction

An array is a collection of more than one homogenous element. The data items /elements of the same type and same length are the homogenous elements. To represent the heterogeneous elements under a single name, an array cannot be used. The solution for this problem is Structures. It is a derived data-type in C. Structure is a collection of different data type elements with different lengths.. It is a convenient tool for handling a group of logically related data items. Structure helps to organize the complex data in more meaningful way.

1.2 Defining a Structure

To use the structure within a program, the special data type, called as structured data type has to be created.. The general format for defining the structured data type is given below.

```
Storage class struct Userdefined _variable name
{
    data type member 1;
    data type member 2;
    -----
    -----
    data type member n;
};
```

Here storage class is optional. The *struct* is a keyword. user defined variable name, are same as other variable name. Data type and members are any valid C data objects such as int, float, char. Once the structure is defined it can be used. The format is called as the template. Remember, this template is terminated by a semicolon. The keyword struct declares a structure to hold the details of different data elements. For example:

```
struct book
{
    char title[20];
    char author[15];
    int pages;
    float price;
};
```

Here, the fields: title, author, pages and price are called as structure elements or members. Each of these members belongs to different data types. The name of structure 'book' is called structure tag.

1.3 Declaring Structure Variables

Members of the structure cannot be accessed directly. To access the member of a structure within a program, a variable has to be declared. To declare a structured variable, the following format is used. It includes following elements:

1. The keyword struct
2. The structure tag name
3. List of variable names separated by commas
4. A terminating semicolon

For example, the statement:

```
struct book book1, book2, book3;
```

This declares book1, book2 and book3 as variables of type struct book. The entire structure with variable is given below.

```
struct book
{
    char title[20];
    char author[15];
    int pages;
    float price;
};
struct book book1, book2, book3;
```

The other way to declare a structure variable is to combine both the structure definition and variables declaration in one statement. This is as shown below.

```
struct book
{
    char title[20];
    char author[15];
    int pages;
    float price;
} book1, book2, book3;
```

1.4 Accessing Structure Members

The individual members of a structure can be accessed through the structure variable only. The link between a member and a variable is established through the operator '.' is called as the dot operator or member operator or period operator. The syntax is

Structure variable. member name
For example,

```
struct book
{
    char title[20];
    char author[15];
    int pages;
    float price;
} book1, book2, book3;
book1.price;
book2.author;
```

```
book3.pages;
book1.title;
```

Note that it is possible to access all the members, through a single variable. There is no one to one correspondence between the the number of members to the number of variables. Any variable can access any member of the struct.

1.4.1 Assigning Values to the Members

Members of the structure can be assigned the values as given below.

```
strcpy(book1.title, " Java Programming");
strcpy(book1.author, "patric");
book1.pages = 375;
book1.price = 275.00;
```

scanf can also be used to give values through the keyboard.

```
scanf("%s", book1.title);
scanf("%s", book1.author);
scanf("%d", &book1.pages);
```

1.4.2 Structure Initialization

Like any other data type, a structure variable can be initialized at compile time. The general format for structure initialization is

static structured datatype structured variable = { val1, val2,, valn};
 where val1, Val2 ,..., Valn are the values of the members , member 1, member 2 ,
 ...member n respectively. The values are separated by commas.. Consider the following example.

```
struct time
{
int hrs;
int mins;
int secs;
} t1, t2;

struct time t1 = { 4, 52, 29};
struct time t2 = { 10, 40, 21};
```

This assigns value 4 to t1.hrs, 52 to t1.mins, 29 to t1.secs and value 10 to t2.hrs, 40 to t2.mins, 21 to t2.secs. There is one-to-one correspondence between the members and their initializing values. C does not permit the initialization of individual structure members within the

template. The initialization must be done only in the declaration of actual variables.

Basic functioning of the structure can be well understood by the following programs.

```
1)          #include<stdio.h>
            main()
            {
            struct student
            {
                int roll_no;
                char name[10];
                int age;
            }s;
            printf("Enter roll, name and age: ");
            scanf("%d %s %d", &s.roll_no, s.name, &s.age);
            printf("\nEntered information: \n");
            printf("Roll number: %d", s.roll_no);
            printf("\nName: %s", s.name);
            printf("\nAge: %d",s.age);
            }
```

Result:

Input:

Enter roll, name and age:09kbt657, shyam,

Output:

Input information:

Roll number: 09kbt657

Name: shyam

age: 17

```
2)          #include <stdio.h>
            int main (void)
            {
            struct date
            {
                int month;
                int day;
                int year;
            };
            struct date today;
            today.month = 4;
            today.day = 25;
            today.year = 2009;
```

```

printf ("Today's date is %i / %i / %.2i.\n",
today.month, today.day, today.year % 100);
return 0;
}

```

Output

Today's date is 4/25/09.

3) Write a program to create a book structure having name, author, page and price.

```

#include <stdio.h>
void main()
{
    struct book
    {
        char name[20];
        char auth[20];
        int page;
        float price;
    };
    struct book b;
    printf("\n ENTER THE NAME OF THE BOOK: ");
    gets(b. name);
    printf("\n ENTER THE NAME OF THE AUTHOR: ");
    gets(b. auth);
    printf("\n ENTER THE NUMBER OF PAGES: ");
    scanf("%d", &b.page);
    printf("\n ENTER THE PRICE OF THE BOOK: ");
    scanf("%f", &b.price);
    printf("\n NAME OF THE BOOK: %s",b.name);
    printf("\n NAME OF THE AUTHOR: %s ", b.auth);
    printf("\n NUMBER OF PAGES: %d ", b.page);
    printf("\n PRICE OF THE BOOK: %0.2f ",b.price);
    getch();
}

```

Result:

Input

```

ENTER THE NAME OF THE BOOK: Complete Reference Java
ENTER THE NAME OF THE AUTHOR: Schildt
ENTER THE NUMBER OF PAGES: 1000
ENTER THE PRICE OF THE BOOK: 395

```

NAME OF THE BOOK: Complete Reference Java

NAME OF THE AUTHOR: Schildt

NUMBER OF PAGES: 1000

PRICE OF THE BOOK: 395

1.5 Questions

- 1) What is a structure?
- 2) How structure is different from array?
- 3) How to declare a structure? explain with an example
- 4) How do you access the members of structure? Explain with an example.
- 5) How do you initialize the structure ? Explain with an example.
- 6) Define a structure that contains the following three members:
 - (a) an integer quantity called won
 - (b) an integer quantity called l o s t
 - (c) a floating-point quantity called percentage Include the user-defined data type record within the definition)

1.6 Programing Exercise

- 1) Develop a program in c using structures to read the following information from the keyboard.
employee name
employee code
designation
age
- 2) Write a program to assign some values to structure members. Display it on the screen using the structure tag.
