SRINIVAS UNIVERSITY

Mukka – 574146, Mangalore, India, Phone: 0824-2477456
(State Private University Established by Karnataka Govt. Act No. 42 of 2013 empowered to award degrees under Section 22 Of UGC Act of UGC, New Delhi, & Member of Association of Indian Universities, New Delhi)
Web: www.srinivasuniversity.edu.in, Email: info@srinivasuniversity.edu.in

PROBLEM SOLVING USING C STUDY METERIAL

For

Ist Semester Bachelor of Computer Application (BCA)

(2021-22)

INSTITUTE OF COMPUTER SCIENCE & INFORMATION SCIENCES

SYLLABUS

Paper Code: 21CAC-1/SD **Theory/Week:** 3 Hours **Credits:** 3	**PROBLEM SOLVING USING C**	**Hours:** 40 **IA :** 50 **Exam:** 50

Course Objectives:

- To describe the basics of computer program and understand the problem-solving aspect.
- To demonstrate the algorithm and flow chart for the given problem.
- To introduce students to the basic knowledge of programming fundamentals of C language.
- To impart writing skill of C programming to the students and solving problems.
- To impart the concepts like looping, array, functions, pointers, file, structure

Course Outcomes:

After the completion of the course students will be able

CO1: To learn and relate the basics of problem-solving methods by writing algorithms and flowcharts.

CO2: To demonstrate how to convert algorithms and flowchart in to C Program.

CO3: To develop coding skills by solving simple and complex problems.

Module – I : 8hrs

Problem Solving techniques: Introduction, Problem solving procedure, Algorithm: Steps involved in algorithm development, Algorithms for simple problems: To find largest of three numbers, factorial of a number, check for prime number, check for palindrome , Count number of odd, even and zeros in a list of integers. Flowcharts: Definition, advantages, Symbols used in flow charts. Flowcharts for simple problems mentioned in algorithms. Psuedocode.

Introduction to C: Overview of C Program, Importance of C Program, Basic structure of a C-program, Execution of C Program.

Constants, Variables and Data types: Character set, C token, Keywords and identifiers, Constants, Variables, data types, Declaration of variables, assigning values to variables, defining symbolic constants.

Teaching Methodology:

Chalk and Board: Introduction, Problem solving procedure, Algorithm: Steps involved in algorithm development Algorithms for simple problems: To find largest of three numbers, factorial of a number, check for prime number , check for palindrome , Count number of odd, even and zeros in a list of integers. Flowcharts: Definition, advantages, Symbols used in flow charts. Flowcharts for simple problems mentioned in algorithms. Psuedocode.

Power Point presentation: Overview of C Program, Importance of C Program, Basic structure of a C-program, Character set, C token, Keywords and identifiers, Constants, Variables, data types, Declaration of variables, assigning values to variables, defining symbolic constants.

Activity based Teaching: Algorithm and flow chart to Count number of odd, even and zeros in a list of integers.

Laboratory Exercise: Execution of C Program. C program to find largest of three numbers, factorial of a number, check for prime number , check for palindrome , Count number of odd, even and zeros in a list of integers.

Module – II: 8hrs

Operators and Expression: Arithmetic, Relational, logical, assignment, increment and decrement, conditional, bitwise and special operators, evaluation of expressions, Precedence of arithmetic operators, type conversions in expressions, operator precedence and Associativity, built in mathematical functions. Managing Input and Output operations: Reading and writing a character, formatted input and output

Decision Making and Branching: Decision making with if statement, simple if statement, the if else statement, nesting of if … else statements, the else if ladder, the switch statement, the ?: operator, the go to statement.

Teaching Methodology:

Chalk and Board: Arithmetic, Relational, logical, assignment, increment and decrement, conditional, bitwise and special operators, Reading and writing a character, formatted input and output.

Power Point presentation: Evaluation of expressions, Precedence of arithmetic operators, type conversions in expressions, built in mathematical functions. Decision making with if statement, simple if statement, the if else statement, nesting of if … else statements, the else if ladder, the switch statement, the ?: operator, the go to statement.

Self Study Material: operator precedence and Associativity.

Laboratory Exercise: Working examples of input and output functions, Decision making with if statement, simple if statement, the if else statement, nesting of if … else statements, the else if ladder, the switch statement, the ?: operator, the go to statement.

Module – III: 8hrs

Decision making and looping: The while statement, the do statement, for statement, exit, break, jumps in loops.

Arrays: Declaration, initialization and access of one-dimensional and two-dimensional arrays. Programs using one- and two-dimensional arrays, sorting and searching arrays.

Handling of Strings: Declaring and initializing string variables, reading strings from terminal, writing strings to screen, Arithmetic operations on characters, String Handling functions, table of strings.

Teaching Methodology:

Chalk and Board:The while statement, the do statement, for statement, exit, break, jumps in loops. Declaring and initializing string variables, reading strings from terminal, writing strings to screen, Arithmetic operations on characters, String Handling functions, table of strings.

Power Point presentation: Declaration, initialization and access of one-dimensional and two-dimensional arrays. Programs using one- and two-dimensional arrays.

Activity based Teaching: sorting and searching arrays.

Video Lectures: Arrays and Strings in C Programming Language.

Laboratory Exercise: Illustration examples of while, do, for statements. Sorting and searching in arrays.

Module – IV: 8hrs

User-defined functions: Need for user-defined functions, Declaring, defining and calling C functions, return values and their types, Categories of functions: With/without arguments, with/without return values. Nesting of functions.

Recursion: Definition, example programs.

Storage classes: The scope, visibility and lifetime of variables.

Teaching Methodology:

Chalk and Board: Need for user-defined functions, The scope, visibility and lifetime of variables.

Power Point presentation: Declaring, defining and calling C functions, return values and their types, Categories of functions: With/without arguments, with/without return values. Nesting of functions.

Activity based Teaching: Recursion, Definition, example programs.

Video Lectures: User defined and built-in functions in C
Laboratory Exercise: User defined functions-Illustrations.

Module – V: 8hrs

Structures and unions: Structure definition, giving values to members, structure initialization, comparison of structure variables, arrays of structures, arrays within structures, Structure and functions, structures within structures. Unions.
Pointers: Understanding pointers, accessing the address of a variable, declaring and initializing pointers, accessing a variable through its pointer, pointer expression, pointer increments and scale factor, pointers and arrays, pointer and strings, passing pointer variables as function arguments.
File Management: Create in Read/Write and Append mode, copying file.

Teaching Methodology:
Chalk and Board: Structure definition, giving values to members, structure initialization, Unions. Understanding pointers, accessing the address of a variable, declaring and initializing pointers, accessing a variable through its pointer.
Power Point presentation: comparison of structure variables, arrays of structures, arrays within structures, Structure and functions, structures within structures. pointer expression, pointer increments and scale factor, pointers and arrays, pointer and strings, passing pointer variables as function arguments.
Video Lectures: Structures and Unions in C Programming Language
Laboratory Exercise: Illustration of example programs in structure and pointers.

PROBLEM SOLVING USING C
(Teaching Plan)

Total Hours: 40

Module 1: Problem Solving techniques

Session1: Introduction, Problem-solving procedure

Session2: Algorithm: Steps involved in algorithm development, Algorithms for simple problems: To find the largest of three numbers, factorial of a number

Session3: check for prime number, check for palindrome ,Count number of odd, even and zeros in a list of integers.

Session4: Flowcharts: Definition, advantages, Symbols used in flow charts.

Session5: Flowcharts for simple problems mentioned in algorithms.
Psuedocode.

Session6: Introduction to C: Overview of C Program, Importance of C Program, Basic structure of a C-program, Execution of C Program.

Session7: Constants, Variables and Data types: Character set, C token, Keywords and identifiers, Constants, Variables, data types

Session8: Declaration of variables, assigning values to variables, defining symbolic constants.

Module 2: Operators and Expression

Session1: Arithmetic, Relational, logical, assignment, increment and decrement, conditional, bitwise and special operators

Session2: evaluation of expressions, Precedence of arithmetic operators, type conversions in expressions

Session3: operator precedence and Associativity, built in mathematical functions.

Seesion4: Managing Input and Output operations: Reading and writing a character, formatted input and output

Session5: Decision Making and Branching: Decision making with if statement, simple if statement

Session6: the if else statement, nesting of if … else statements

Session7: the else if ladder, the switch statement,

Session8: the ?: operator, the go to statement.

Module 3: Decision making and looping

Session1: The while statement, the do statement

Session2: for statement, exit, break, jumps in loops.

Session3: Arrays: Declaration, initialization and access of one-dimensional and two-dimensional arrays.

Session4: Programs using one- and two-dimensional arrays,

Session5: sorting and searching arrays.

Session6: Handling of Strings: Declaring and initializing string variables, reading strings from terminal

Session7: writing strings to screen, Arithmetic operations on characters

Session8: String Handling functions, table of strings.

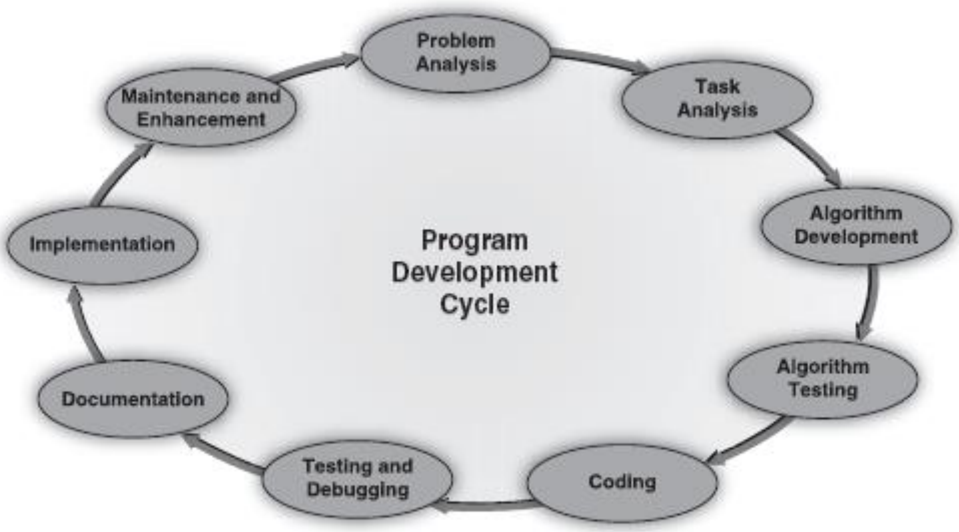
Module 4: User-defined functions:
Session1: Need for user-defined functions, Declaring, defining and calling C functions,
Session2: return values and their types
Session3: Categories of functions: With/without arguments
Session4: with/without return values.
Session5: Nesting of functions.
Session6: Recursion: Definition, example programs.
Session7: Storage classes: The scope
Session8: visibility and lifetime of variables.

Module 5: Structures and unions, Pointers, File Management:
Session1: Structure definition.
Session2: giving values to members, structure initialization, comparison of structure variables
Session3: Arrays of structures, arrays within structures
Session4: Structure and functions, structures within structures.
Session5: Unions.
Session6: Understanding pointers, accessing the address of a variable**.**
Session7: declaring and initializing pointers**,** accessing a variable through its pointer.
Session8: pointers and arrays, pointer and strings.

UNIT 1
CHAPTER 1: PROBLEM SOLVING TECHNIQUES

Program Development Cycle

the process of writing a program (coding), the programmer has to determine the problem that needs to be solved. There are different approaches to problem solving. One common approach is to use the *program development cycle*.



- **Problem Analysis:** The problem is analysed precisely and completely. Based on understanding, the developer knows about the scope within which the problem needs to be developed.
- **Task Analysis:** After analysing the problem, the developer needs to develop various solutions to solve the given problem. From these solutions, the optimum solution is chosen, which can solve the problem comfortably and economically.
- **Algorithm Development:** After selecting the appropriate solution, an algorithm is developed to depict the basic logic of the selected solution. An algorithm depicts the solution in logical steps (sequence of instructions). Further, an algorithm is represented by flowcharts, decision tables and pseudocodes (informal way of programming description). These tools make the program logic clear and they eventually help in coding.
- **Algorithm Testing:** Before converting the algorithms into actual code, it should be checked for accuracy. The main purpose of checking the algorithm is to identify major logical errors at an early stage because logical errors are often difficult to detect and correct at later stages. The testing also ensures that the algorithm is a "true" one and it should work for both normal as well as unusual data.
- **Coding:** After meeting all the design considerations, the actual coding of the program takes place in the chosen programming language. Depending upon the application domain and available resources, a program can be written by using

computer languages of different levels such as machine, assembly or high-level languages (HLL).

- **Testing and Debugging:** It is common for the initial program code to contain errors. A program compiler and programmer-designed test data machine tests the code for syntax errors. The results obtained are compared with results calculated manually from these test data. Depending upon the complexity of the program, several rounds of testing may be required.
- **Documentation:** Once the program is free from all the errors, it is the duty of the program developers to ensure that the program is supported by suitable documentation. These documents should be supplied to the program users. Documenting a program enables the user to operate the program correctly. It also enables other persons to understand the program clearly so that it may, if necessary, be modified, or corrected by someone other than the original programmer.
- **Implementation:** After documentation, the program is installed on the end user's machine and the user is also provided with all the essential documents in order to understand how the program works. The implementation can be viewed as the final testing because only after using the program the user can point out the drawbacks (if any) and report them to the developers. Based on the feedback from users, the programmers can modify or enhance the program.
- **Maintenance and Enhancement:** After the program is implemented, it should be properly maintained by taking care of the changing requirements of its users and the system. The program should be regularly enhanced by adding additional capabilities. This phase is also concerned with detecting and fixing the errors, which were missed in the testing phase. Since this step generates user feedback, the programming cycle continues as the program is modified or reconstructed to meet the changing needs.

ALGORITHM

Algorithms are one of the most basic tools that are used to develop the problem-solving logic. An *algorithm* is defined as a finite sequence of explicit instructions that when provided with a set of input values produces an output and then terminates

Algorithm Properties

Algorithms are not computer programs, as they cannot be executed by a computer. Some properties of algorithm are as follows.

- There must be no ambiguity in any instruction.
- There should not be any uncertainty about which instruction is to be executed next.
- The description of the algorithm must be finite. An algorithm cannot be open ended (there should be some limit).
- The algorithm should terminate after a finite number of steps.

- The algorithm must be general enough to deal with any contingency (possible but cannot be predicted).

Advantages of algorithm

- it is a step-by-step representation of a solution to a given problem ,which is very easy to understand
- it has got a definite (clearly stated or decided) procedure.
- it easy to first develop an algorithm, then convert it into a flowchart &then into a computer program.
- it is independent of programming language.
- it is easy to debug as every step is got its own logical sequence

Disadvantages of algorithm

It is time consuming & cumbersome (complicated) as an algorithm is developed first which is converted into flowchart &then into computer program

Type of Algorithms :

The algorithm and flowchart, classification to the three types of *control structures. They are:*

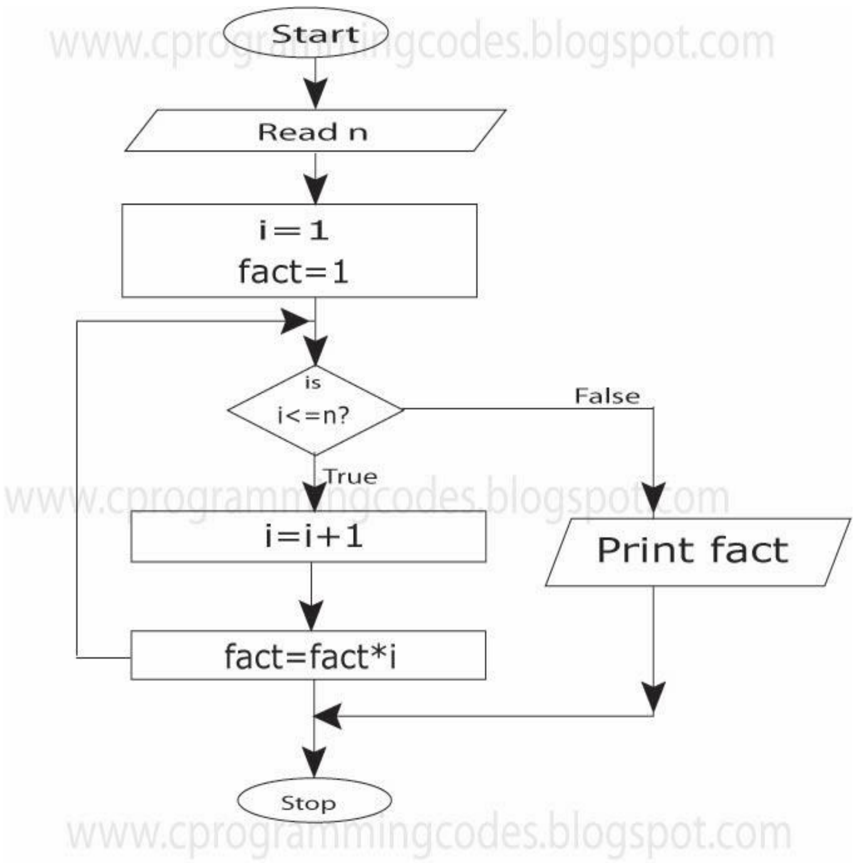
1. Sequence (particular order)
2. Branching (Selection)
3. Loop (Repetition)

Q)EXAMPLE:Write algorithm to calculate the sum and average of two numbers.

```
Step1 : Start
Step2 : Read two numbers n,m
Step3 : Calculate sum=n+m
Step4 : Calculate avg=sum/2
Step5 : Print sum,avg
Step 6 : Stop
```

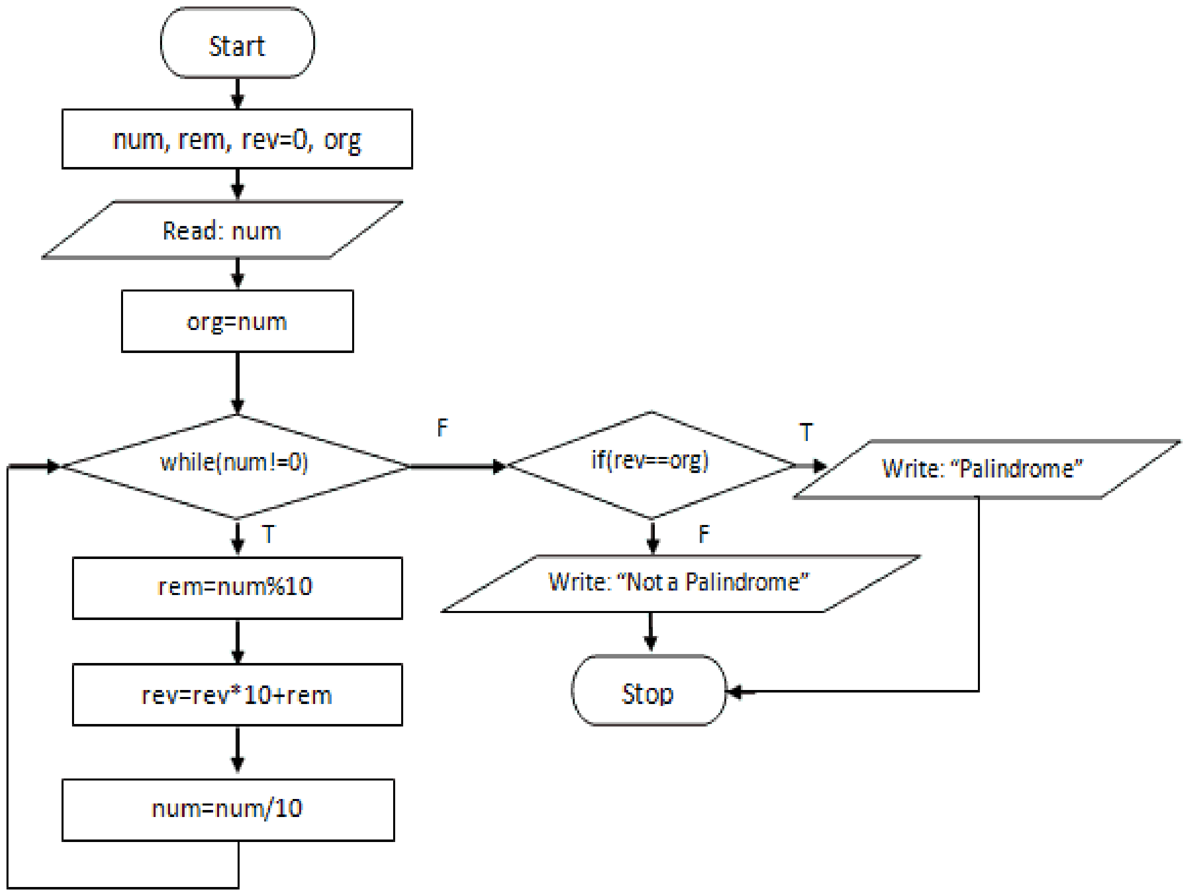
Q)WRITE AN ALGORITHM TO FIND FACTORIAL OF N NUMBER

```
Step 1:start
Step 2: set i=1,fact=1,n=5
Step 3:while(i<=n)
        (a)fact=fact*i
        (b)i=i+1
   [end loop]
Step 4:stop
```



Q)WRITE AN ALGORITHM CHECK FOR PALINDROME AND DRAW FLOWCHART

```
Step 1:start
Step 2:read a number num
Step 3 :set  rev=0,rem=0
Step 4:while num>0 true continue else goto step 8
Step 5 :set rem=num%10
Step 6:set rev=rev*10+rem
step 7:set num=num/10 go to step 4
Step 8:print rev
Step 9:stop
```



Q) Write algoritham to find largest among three number AND FLOWCHART

Step 1;start

Step 2:read a,b,c

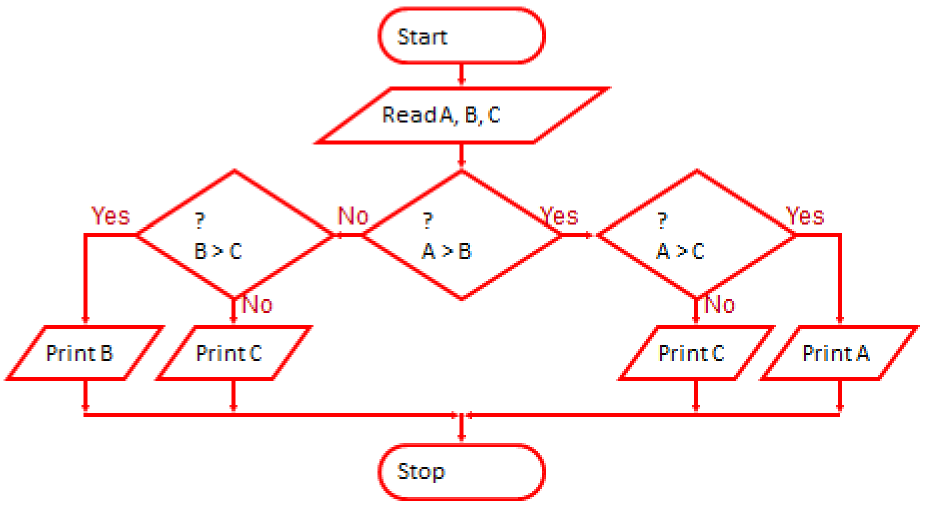
Step 3:if(a>b)and(a>c)then print a goto step 6 else goto step4

Step 4:if(b>c) then print b and goto step 6

else goto step 5

Step 5:print c

Step 6:stop



Q)WRITE AN ALGORITHM TO CHECK GIVEN NUMBER IS PRIME OR NOT

```
Step 1:start
Step2:[read a number] N
STEP 3:IF(N<=3)THEN
PRINT(NUMBER IS PRIME NO)
        GOTO STEP 8
STEP 4:SET  I=2,FLAG=0,
STEP 5:REPEAT STEP 6    WHILE(I<N)
 STEP 6:IF(N%I==0),THEN
                  FLAG=1
         BREAK   ///OUT OF LOOP
ELSE
        I=I+1;
[END IF]
[END LOOP]
STEP 7:IF(FLAG==1)THEN
PRINT(N  NOT IS PRIME NUMBER)
         ELSE
PRINT(N IS  PRIME NUMBER)
STEP 8:STOP
             OR
Check given number is prime or not
```

Step-1: start

Step-2: Read a "n" value to check prime or not

Step 3: set i=1, count=0.

Step 4: if i <=n if true go to step 5, else go to step 8

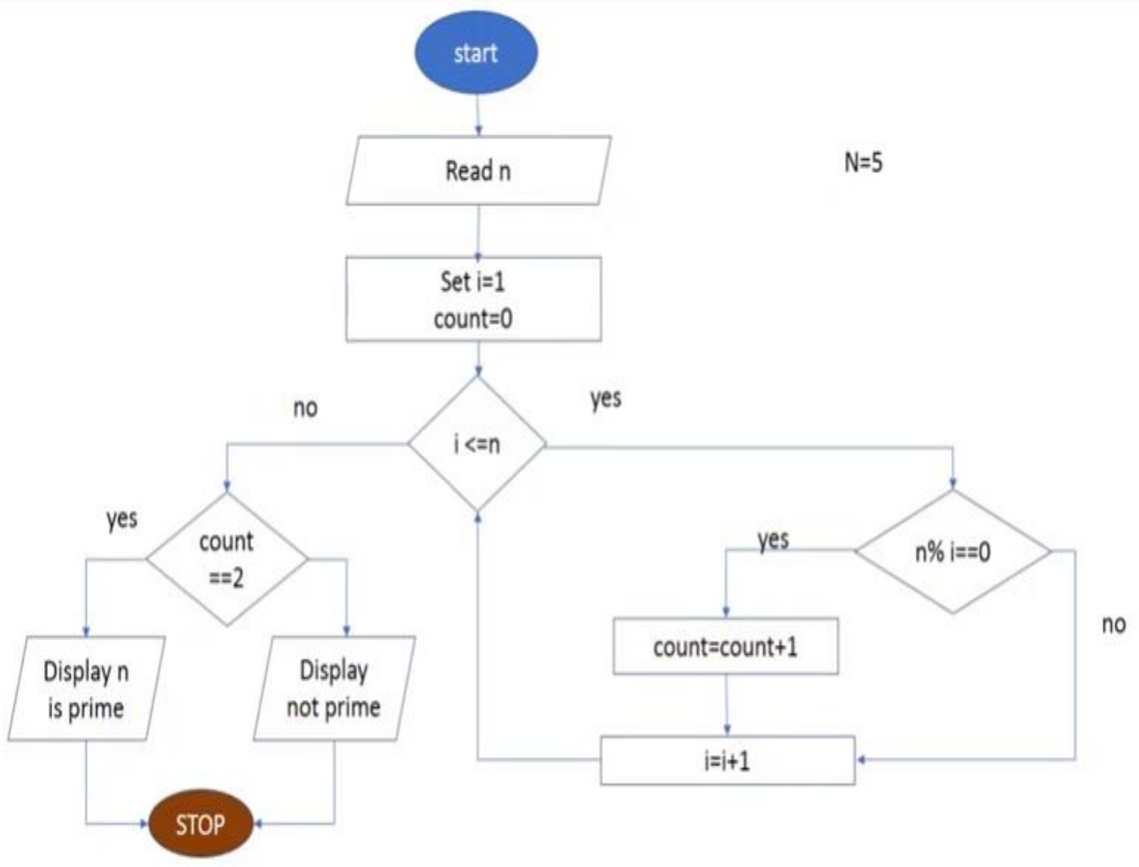
Step 5: check the condition n%i==0 if true then evaluate step 6, false go to step 7.

Step 6: set count =count+1

Step 7: i=i+1 go to step 4

Step 8: check count, if count =2 display prime , if not display it is not prime

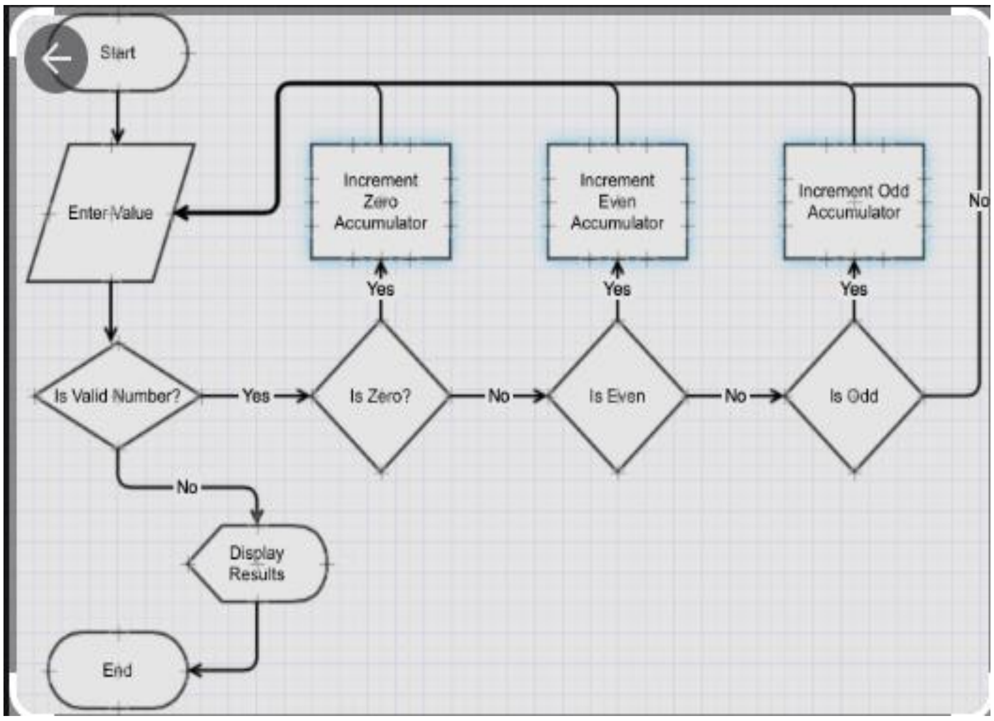
Step 9: Stop



WRITE AN ALGORITHM TO COUNT NUMBER OF ODD, EVEN AND ZEROS IN A LIST OF INTEGERS.

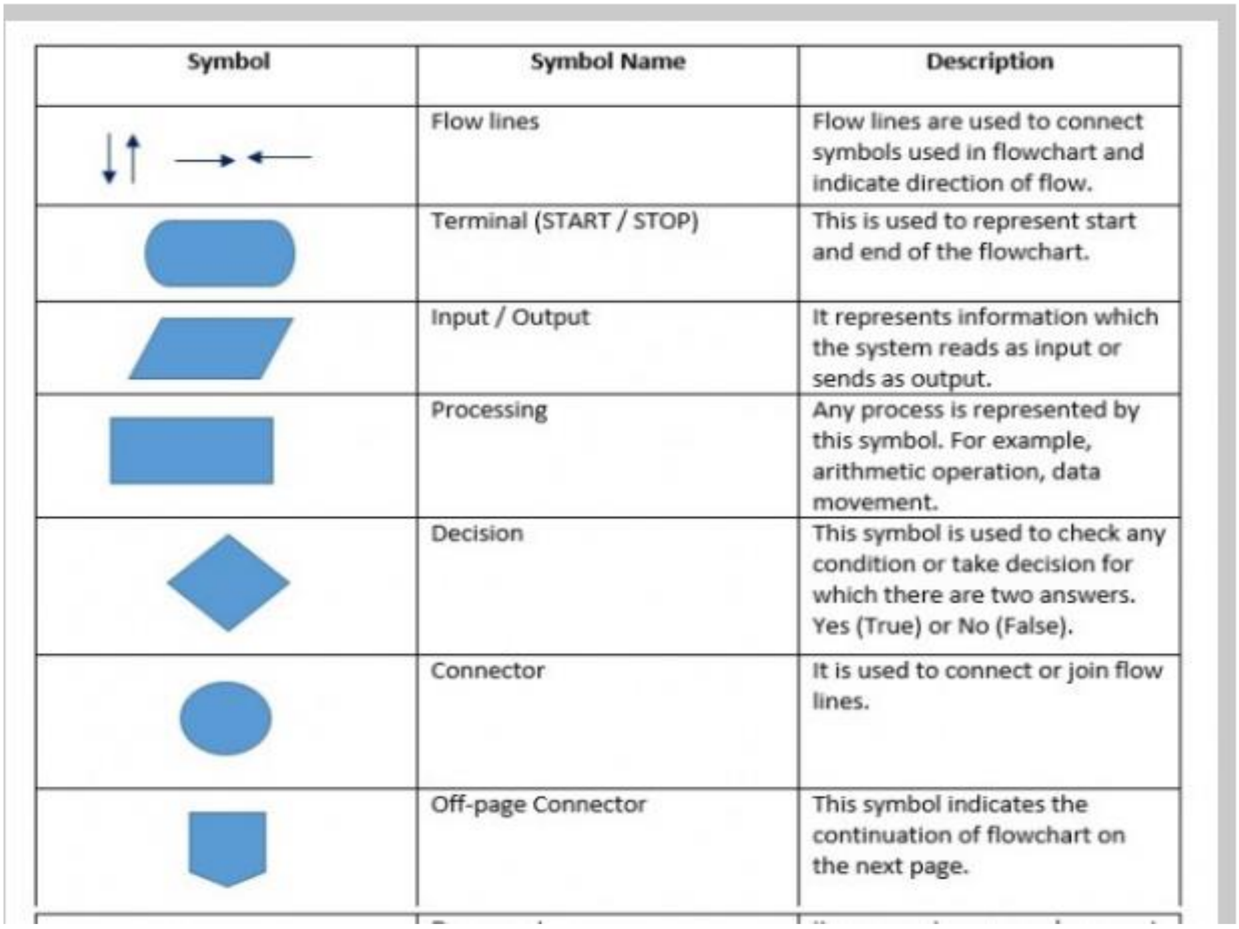
- Step1:Input the number of elements of the array.
- Step2:Input the array elements.
- Step3:Initialize count_zero=count_odd = count_even = 0.
- Step4:Traverse the array and increment count_zero if the array elemnt is equal to zero, increment count_odd if the array element is odd, else increment count_even.

- Step5: Print count_zero, count_odd and count_even.

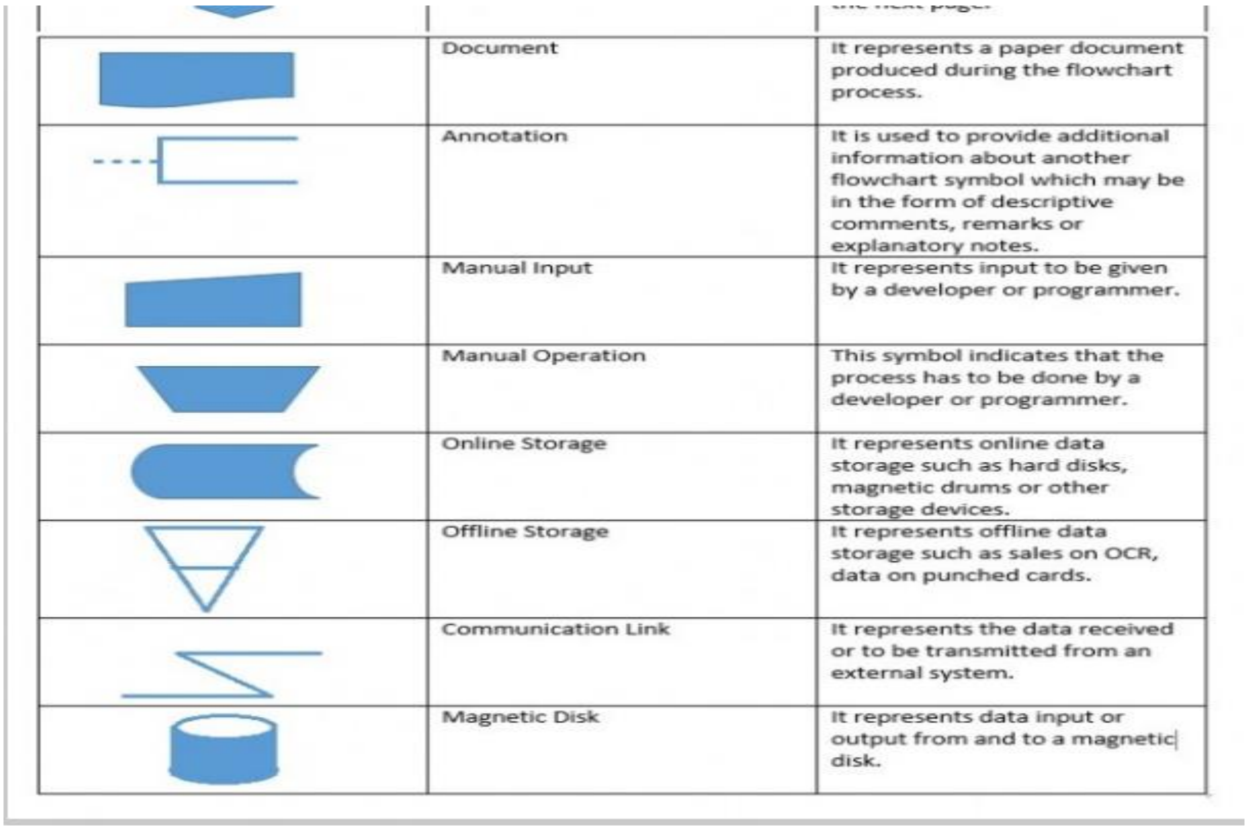


Flowchart:

- 1. Graphical representation of any program is called flowchart.
  2. There are some standard graphics that are used in flowchart as following



Symbol	Symbol Name	Description
↓↑ → ←	Flow lines	Flow lines are used to connect symbols used in flowchart and indicate direction of flow.
⬭	Terminal (START / STOP)	This is used to represent start and end of the flowchart.
▱	Input / Output	It represents information which the system reads as input or sends as output.
▭	Processing	Any process is represented by this symbol. For example, arithmetic operation, data movement.
◆	Decision	This symbol is used to check any condition or take decision for which there are two answers. Yes (True) or No (False).
●	Connector	It is used to connect or join flow lines.
⬟	Off-page Connector	This symbol indicates the continuation of flowchart on the next page.



		the next page.
	Document	It represents a paper document produced during the flowchart process.
	Annotation	It is used to provide additional information about another flowchart symbol which may be in the form of descriptive comments, remarks or explanatory notes.
	Manual Input	It represents input to be given by a developer or programmer.
	Manual Operation	This symbol indicates that the process has to be done by a developer or programmer.
	Online Storage	It represents online data storage such as hard disks, magnetic drums or other storage devices.
	Offline Storage	It represents offline data storage such as sales on OCR, data on punched cards.
	Communication Link	It represents the data received or to be transmitted from an external system.
	Magnetic Disk	It represents data input or output from and to a magnetic disk.

ADVANTAGES OF FLOWCHART

- **Makes Logic Clear:** The main advantage of using a flowchart to plan a task is that it provides a pictorial representation of the task, which makes the logic easier to follow. The symbols are connected in such a way that they show the movement (flow) of information through the system visibly. The steps and how each step is connected to the next can be clearly seen. Even less experienced personnel can trace the actions represented by a flowchart, that is, flowcharts are ideal for visualizing fundamental control structures employed in computer programming.
- **Communication:** Being a graphical representation of a problem-solving logic, flowcharts are a better way of communicating the logic of a system to all concerned. The diagrammatical representation of logic is easier to communicate to all the interested parties as compared to actual program code as the users may not be aware of all the programming techniques and jargons.
- **Effective Analysis:** With the help of a flowchart, the problem can be analysed in an effective way. This is because the analysing duties of the programmers can be delegated to other persons, who may or may not know the programming techniques, as they have a broad idea about the logic. Being outsiders, they often tend to test and analyse the logic in an unbiased manner.
- **Useful in Coding:** The flowcharts act as a guide or blueprint during the analysis and program development phase. Once the flowcharts are ready, the programmers can plan the coding process effectively as they know where to begin and where to

end, making sure that no steps are omitted. As a result, error-free programs are developed in HLL and that too at a faster rate.

- **Proper Testing and Debugging:** By nature, a flowchart helps in detecting the errors in a program, as the developers know exactly what the logic should do. Developers can test various data for a process so that the program can handle every contingency.
- **Appropriate Documentation:** Flowcharts serve as a good program documentation tool. Since normally programs are developed for novice users, they can take the help of the program documentation to know what the program actually does and how to use the program.

Limitations of Flowcharts

- **Complex:** The major disadvantage in using flowcharts is that when a program is very large, the flowcharts may continue for many pages, making them hard to follow. Flowcharts tend to get large very quickly and it is difficult to follow the represented process. It is also very laborious to draw a flowchart for a large program. You can very well imagine the nightmare when a flowchart is to be developed for a program, consisting of thousands of statements.
- **Costly:** Drawing flowcharts are viable only if the problem-solving logic is straightforward and not very lengthy. However, if flowcharts are to be drawn for a huge application, the time and cost factor of program development may get out of proportion, making it a costly affair.
- **Difficult to Modify:** Due to its symbolic nature, any changes or modification to a flowchart usually requires redrawing the entire logic again, and redrawing a complex flowchart is not a simple task. It is not easy to draw thousands of flow lines and symbols along with proper spacing, especially for a large complex program.
- **No Update:** Usually programs are updated regularly. However, the corresponding update of flowcharts may not take place, especially in the case of large programs. As a result, the logic used in the flowchart may not match with the actual program's logic. This inconsistency in flowchart update defeats the main purpose of the flowcharts, that is, to give the users the basic idea about the program's logic.

PSEUDOCODE

Pseudocode

***Pseudocode* is made up of two words: *pseudo* and *code*. Pseudo means imitation and code refers to instructions, written in a programming language. As the name suggests, pseudocode is not a real programming code, but it models and may even look like programming code**. It is a generic way of describing an algorithm without using any specific programming language-related notations. Simply put, pseudocode is an outline of a program.

Pseudocode uses some keywords to denote programming processes. Some of them are as follows.

- **Input:** READ, OBTAIN, GET and PROMPT

- **Output:** PRINT, DISPLAY and SHOW
- **Compute:** COMPUTE, CALCULATE and DETERMINE
- **Initialize:** SET and INITIALIZE
- **Add One:** INCREMENT

Pseudocode: To Calculate the Area of A Rectangle

```
• PROMPT the user to enter the height of the rectangle
• PROMPT theuser to enter the width of the rectangle
• COMPUTE the area by multiplying the height with width
• DISPLAY the area
• STOP
```

Benefits of Pseudocode

- Since it is language independent, it can be used by most programmers. It allows the developer to express the problem logic in plain natural language.
- It is easier to develop a program from a pseudocode rather than from a flowchart or decision table. Programmers do not have to think about syntaxes; they simply have to concentrate on the underlying logic. The focus is on the steps to solve a problem rather than on how to use the computer language.
- The words and phrases used in pseudocode are in line with basic computer operations. This simplifies the translation from the pseudocode to a specific programming language.
- Unlike flowcharts, pseudocode is compact and does not tend to run over many pages. Its simple structure and readability makes it easier to Modify.

Limitations of Pseudocode

- It does not provide a visual representation of the program's logic.
- There are no accepted standards for writing pseudocodes. Programmers use their own style of writing pseudocode.
- It is quite difficult for the beginners to write pseudocode as compared to drawing a flowchart.

PROGRAM CONTROL STRUCTURES

. Program statements that affect the order in which statements are executed, or that affect whether statements are executed, are called *control structures*. They affect the flow of simulation code since a control structure evaluates statements and then executes code according to the result.

there are three control structures.

- **Sequence:** where information flows in a straight line.

- **Selection:** (decision or branched), where the decisions are made according to some predefined condition.
- **Repetition:** (looping), where the logic (sequence of steps) is repeated in a loop until the desired output is obtained.

Unit 1
Chapter2
Overview of c

HISTORY OF C

- The root of all modern programming language is ALGO, it is introduced in 1960. ALGO gave the new concept of the structure programming. Subsequently varies languages are announced.
- In 1967 Martin Richards developed a language called BCPL(Basic Combined Programming Language) .
- In 1970, Ken Thompson created a language using many features of BCPL and it simply called B language.
- Both BCPL and B was "type less" system programming language.
- C language was developed by Dennis Ritchie at the Bell Laboratories in 1972.
- C language uses many features of the ALGO, BCPL and B Languages. It also added new concepts of the "data type" and many powerful features.
- Unix operating system is totally coded in C language
- In 1983 American National Standard Institute (ANSI) appoint a technical committee to define a standard of C.
- The committee approved a version of c in December 1989 which is now known as ANSI C
- It was the approved by the International Standards Organization (ISO) in 1990.This version of the c is also referred as C89

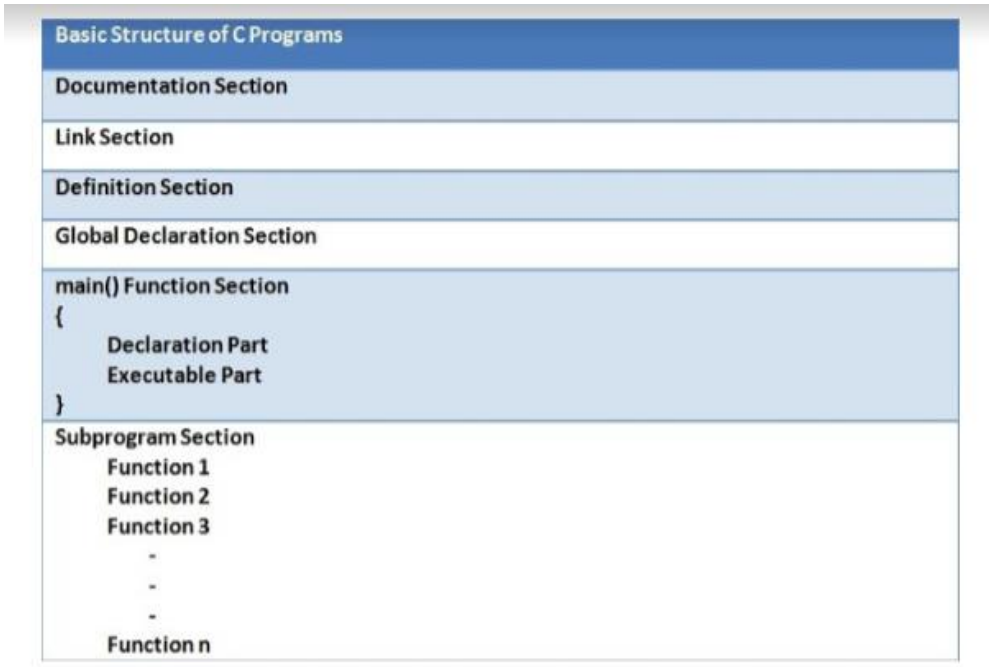
SALIENT FEATURES OF C (IMPORTANCE OF C)

C has many advantages over other high level languages.

- It is robust.
- C has the advantage of assembly level programming such as bit manipulation and all the significant features of high level language such as easy debugging, compactness etc. Therefore most of the C compilers are written in C.
- C is also called as a middle level language since it combines the features of high level as well as low level programming.
- It is highly suited for writing system software and application packages.

- C consists of a rich variety of data types and powerful operators. This makes C programs much more efficient and fast.
- It is platform independent and highly portable i.e., C can be run on almost any operating system.
- C is a structured language, wherein the program is subdivided into a number of modules. Each of these modules performs a specific task. Further structured programming helps in making program debugging, testing and maintenance, easier.
- One of the salient feature of C is its ability to add on to its library. User-defined functions can be added to the C library making the programs simpler.
- C provides manipulation of internal processor registers.
- It allows pointer arithmetic and pointer manipulation.
- Expressions can be represented in compact form using C.

Basic structure of a C program



Basic Structure of C Programs
Documentation Section
Link Section
Definition Section
Global Declaration Section

main() Function Section
{
Declaration Part
Executable Part
}

Subprogram Section Function 1 Function 2 Function 3 - - - Function n

- The **documentation section** consists of a set of comment lines giving the name of the program , the author and other details, which the programmer would like to use later.
- The **link section** provides instructions to the compiler to link functions from the system library
- The **definition section** defines all symbolic constants
- There are some variables that are used in more than one function, such variables are called global variables and are declared in the **global declaration section**, that is outside of all the function. This section also declares all the user defined functions.
- Every Cprogram must have only **one main()** function. It indicates that the name of the function is main and that the program execution should begin here. The empty parenthesis() that follow main indicate that the function has no arguments.

The actual execution of the program begins here.

- This section consists of 2 parts **declaration part and execution part**. All the variables that are used in the program are declared in the declaration part. There must be atleast one statement in the executionpart.
- These 2 parts are enclosed between curly braces{}. Program execution begins at the opening brace { and the closing brace } indicates the logical end of the program.
- Every statement in this Section are terminated by a semicolon(;).
- Every C program must have a main()section.
- The sub program section contains all the user defined functions that are called in the **main function.**

Example:

/* Program to calculate area of a circle */ Document section

```
#include<stdio.h>                    // Link section
#define PI 3.14159                    // Definition section
main()                                // main()function section
{
int radius;
float area;                          // DECLARE VARIABLE
printf("Enter the radius of a circle ");
scanf("%d",&radius);
area=PI*radius*radius;                  // Executable
printf("Area of a circle=%.2f",area);
}
```

Rules for writing a Cprogram:

- Cprogramstatementsmustbewritteninlowercaseletters.Onlysymbolicconstantsar e written in uppercaseletters.
- Every statement should end with asemicolon.
- Bracesareusedtogroupandmark together thebeginningandendoffunctions.
- Properindentationofbracesandstatementsmaketheprogrameasiertoreadanddebug
- C is a case sensitivelanguage.C is a free-form language. i.e. More than one statement can be example: written on one line separated by asemicolon.

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

Can be written on one line as

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

Executing a C program

Executing a program written in C involves series of steps

1. Creating the program
2. Compiling the program
3. Linking the program with functions that are needed from the C library
4. Executing the program

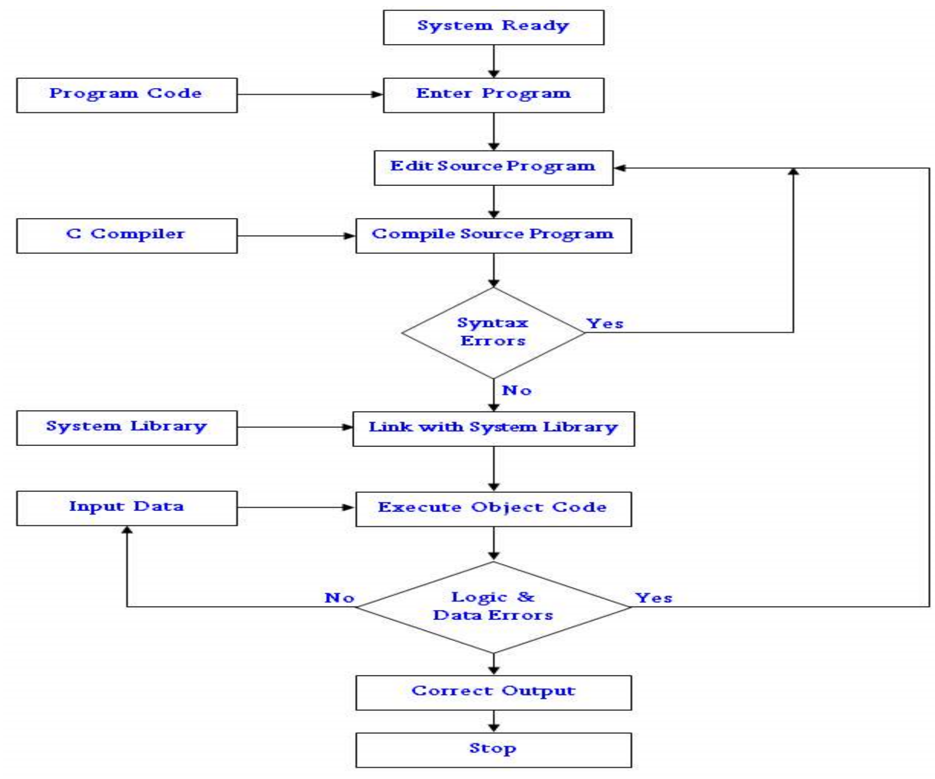


Figure : Process of compiling and running a C program.

The source program translated into suitable execution format. The translation is done after examining each instruction correctness. If everythings all right program translated into **object code.**

Linking is the process of putting together other program files and functions that are required by the program**.**

Example :library function of system linked to main program .the linking is done automatically.

UNIT 1:
CHAPTER 3
CONSTANTS, VARIABLES AND DATA TYPES

Introduction :-

- The Sequence of the Instruction written to perform specific task is called the Program.
- This Instructions are formed using special symbols and words according to some rigid rule is known as syntax rules.
- Every Instruction must be written according to the syntax rule of the languages.
- Like any other languages C languages has its vocabulary and grammar. They are follow up for making the program.

In this Chapter we will discuss about various constants, variables and data types are used in C Programming

Characterset:

The characters that are used to form statements and expressions in a programming language is known as character set. The characters set in C are grouped as follows.

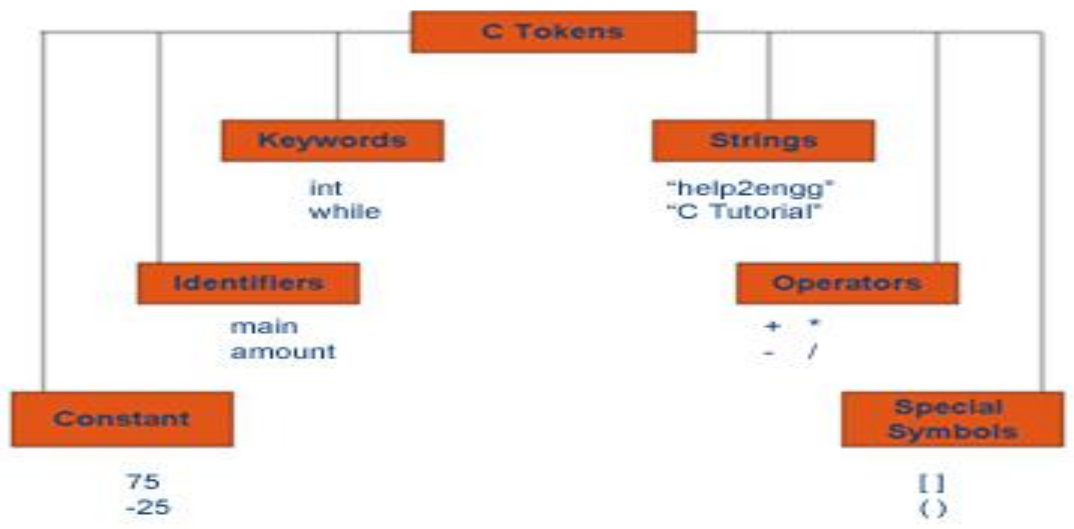
1. Letter- uppercase A-Z and lowercase a-z
2. Digits-0-9
3. Special characters (, > < ? ‘ “ ; :)
4. Whitespaces (blank spce, new line, tab)

C TOKENS

- In a passage of text, individual words and punctuation marks are called tokens,
- In C program the smallest individual units are known as tokens
- C programs are written using these tokens and the syntax of the language

. C supports 6 types of tokens. They are

- Keywords
- Identifiers
- Constants
- Strings
- Specialsymbols
- Operators



Keywords and identifiers:

- Every C word is classified either as a keyword or an identifier
- All keywords have a fixed predefined meaning that cannot be changed. Hence keywords are also known as reserve words.
- All keywords must be written in lowercase.
- C supports a set of 32keywords

Ex: int, float, double, extern, static, auto, continue, if, goto short, long etc. are the keywords

Keywords in C Language

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

Identifiers

Identifiers refer to the names of variables, functions and arrays
These are user defined names and consists of a sequence of letters and digits, with a letter as a first character, both uppercase and lowercase letters are permitted.

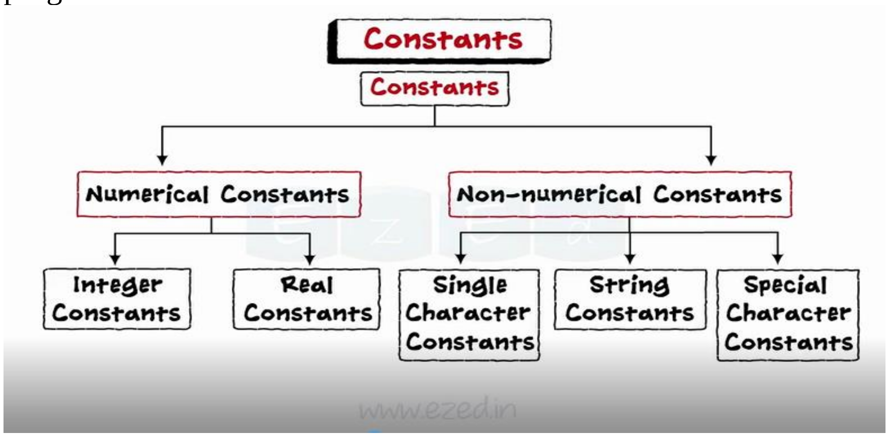
Rules for identifiers

1. First character must be an alphabet (or underscore)
2. Must consist of only letters, digits or underscore.

3 Only first 31 characters are significant
4 Cannot use a keyword
5 Must not contain white space.

Constants

Constants:

- Constants referred as a fixed value that don't change during the execution of the program.
- 

Integer Constants:

Integer constants refer to a sequence of digits preceded by an optional +/- sign.
E.g.+123,-567,

- Decimal integers consist of a set of digits 0-9.
- An octal integer consists of combination of digits from 0 to 7 with a leading **0**. E.g.032,056,0435.
- A hexadecimal integer consists of 0-9 digits and alphabets A through F to represent numbers from 10-15. A hexadecimal integer is preceded by **0x** or 0X. E.g. 0xF2, 0X35A,0x56

Real constants:

- Numbers containing fractional part are called real or floating point constants.
- These can be represented either in decimal notation or exponential notation.
- In decimal notation a whole number is followed by a decimal point and a fractional part. E.g. 0.045,-0.25
- A real number can also be expressed in scientific or exponential notation as mantissa exponent.
- The mantissa could be either a real number expressed in decimal notation or an integer.
- The exponent is an integer with an optional ^/-sign.
- E.g.0.65e4,1.25e-5,56.23E-1

Character constants:

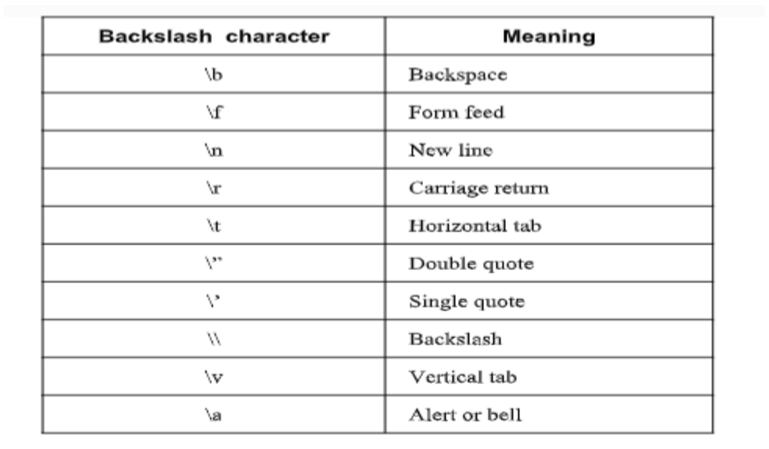
- Character constants are either single character constants or string constants
- Single character constants consist of a single character in single quotes.

E.g.‘a’,‘Q’,‘5’

- String constants are a group of characters enclosed in double quotes. The characters may include letters, numbers or special characters. E.g.“abc”, “1956”,“Hello!”

Backslash character constants:

- C supports backslash character constants that are used in output function.
- A backslash character constant consists of a backslash followed by a character. These character combinations are also known as escape sequences



Backslash character	Meaning
\b	Backspace
\f	Form feed
\n	New line
\r	Carriage return
\t	Horizontal tab
\”	Double quote
\’	Single quote
\\	Backslash
\v	Vertical tab
\a	Alert or bell

Variables:

- A variable is a data name that may be used to store a data value.
- A variable may take different values at different times during execution.
- The value of a variable changes during the execution of program.

Rules for naming a variable:

- Variable names should begin with a letter.
- Variable names should not be keywords.
- Blank spaces are not allowed.
- Uppercase and lowercase letters are significant because C is a case sensitive language.
- Length of a variable name should not exceed 8 characters

DECLARATION OF VARIABLE

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

1. It tells the compiler what the variable name is.
2. It specifies what type of data the variable will hold.
3. The declaration must be done before they are used in the program.

DATA TYPE VAR1,VAR 2,VAR3;

- Variables are separated by commas
- The declaration statement must end with a semicolon

<u>Assigning values toVariables:</u>

- Values are assigned to variables using the assignment operator'='.

Syntax: var-name=value;

E.g.: x=10;

<u>INITIALIZATION</u>

It is also possible to assign values to a variable during declaration. The process of assigning initial values to variables is known as initialization.

Syntax: data type var-name=value;

E.g.: int x=10;
char x='y';

<u>Declaring variable as a constant:</u>

Sometimesinprogramming,wewouldlikevariablevaluesunchangedduringprogram execution.Thiscanbeachievedbydeclaringthevariablewithaqualifierconstantatthe time of initialization.

Defining constant

- Using *#define* preprocessor directive
- Using a *const* keyword

Syntax: const data type var-name=va1ue;

EX: const int x=40;

Difference between constant variable and variable

- The variable whose value does not change during the execution of program is known as *constant variable."*
- a variable may take different value at different time.

<u>DEFINING SYMBOLIC CONSTANTS</u>

A symbolic constant is a name, which substitutes either a numeric constant or a character or a string constant. When the program is compiled, each occurrence of a symbolic constant is replaced by its corresponding character sequence. Symbolic constants are usually defined at the beginning of a program.

Symbolic constant can be defined using **#define statement.**

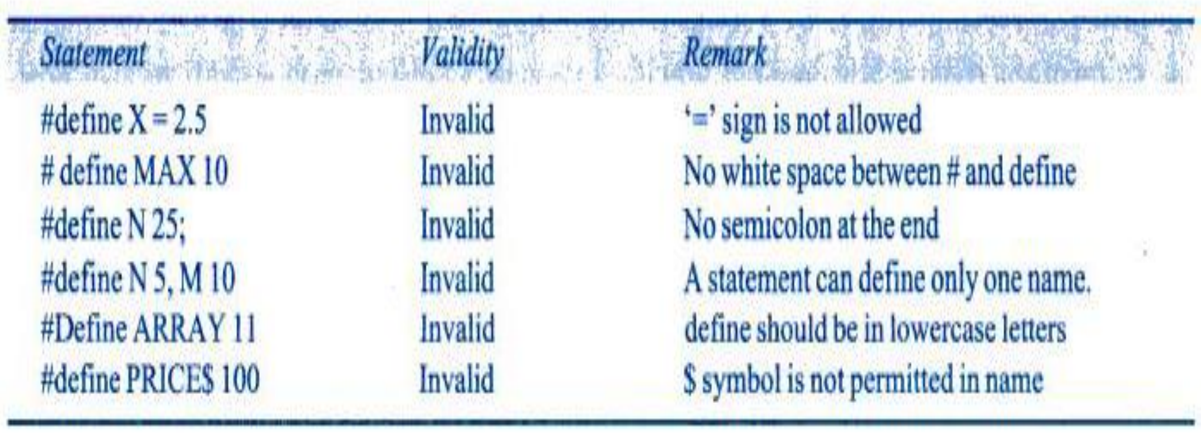
#define symbolic-name value-of constant

Ex:-

- #define PI 3.142
- #define MAX 100
- #define CLASS "IBCA"

Following are the rules used to define the symbolic constant:

- Symbolic name is similar to variable but it is not variable.
- No blank space is allow between '#' and 'define' word.
- '#' must be first in the line.
- A blank space is require between '#define' and 'symbolic name' & between 'symbolic name' and 'constant value'
- It is not statement so it does not end with semicolon
- You cannot put assignment operator '=' between 'symbolic name' and constant value.
- You can put anywhere in program but before it is used in the program.



Statement	Validity	Remark
#define X = 2.5	Invalid	'=' sign is not allowed
# define MAX 10	Invalid	No white space between # and define
#define N 25;	Invalid	No semicolon at the end
#define N 5, M 10	Invalid	A statement can define only one name.
#Define ARRAY 11	Invalid	define should be in lowercase letters
#define PRICE$ 100	Invalid	$ symbol is not permitted in name

Data Types:

- C language is rich in its data types. A variety of data types allow the programmer to select the type according to the application.
- The 4 basic data types supported by C are

1. Fundamental / Primary Data type
2. User defined Data type
3. Derived data type
4. Empty dataset

Primary Data Types:

- All C compilers support this data type. Primary data types can be classified as follows
  1. Integer
  2. Floating point
  3. Character

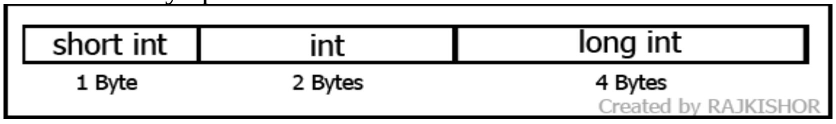
Data type	Range of Values
Char	-128 to +127
Int	-32,768 to +32,767
Float	3.4e-38 to 3.4e+38
Double	1.7e-308 to 1.7e+308

Integer Data Type:

- Integers are whole numbers with a range of values. Integers occupy one word of storage.
- In a 16-bit machine, the range of integers is -32768 to+32767
- In order to provide some control over a range of numbers and storage space, C

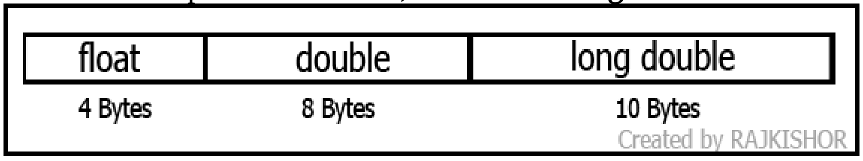
has 3 classes of storage for integers namely, short int, int and long int in both signed and unsigned format

- Short int is used for small range of values and requires half the amount of storage as a regular int number uses.
- Long int requires double the storage as an int value.
- In unsigned integers, all the bits in the storage are used to store the magnitude and are always positive.



Floating point DataType:

- All floating point numbers require 32 bits with 6 digits of precision
- They are defined in C using the keyword float
- For higher precision, double data type can be used.
- A **double type number uses 64 bits (8 bytes) giving a precision of 14 digits.**
- Range of values for double type is **1.7E-308 to 1.7E+308**
- To extend the precision further, we can use **long double which uses 80 bits.**



Character DataType:

- A single character can be defined as a character data type using the keyword char
- Characters usually need 8 bits for storage
- The qualifier **signed or unsigned can be explicitly applied to char.**
- While **unsigned characters have values between 0 and 255,**
- **signed characters have values from -128 to 127.**

Unit 2

CHAPTER 1

OPERATORS AND EXPRESSIONS

C supports a rich set of operators. An operator is a symbol that tells the computer to perform an arithmetic or logical function. Operators are used in programs to manipulate data and variables. Operators in C can be classified into a number of categories. They are

- ✓ ArithmeticOperators
- ✓ RelationalOperators
- ✓ LogicalOperators
- ✓ AssignmentOperators
- ✓ Increment and DecrementOperators
- ✓ ConditionalOperators
- ✓ BitwiseOperator
- ✓ SpecialOperators

ArithmeticOperators:

All the basic arithmetic operators are supported by C. These operators can manipulate with any built in data types supported by C. The various operators are tabulated as follows:

Operator	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo Division

Integer Arithmetic:

When both the operands in the expression are integers, then the operation is known as integer arithmetic.
For example :if the values of a=14 and b=4 then
a+b=18 ,
a-b=10,
a*b=56 ,
a/b=3,

During modulo division, the sign of the result is the sign of the first operand.
E.g. -14%3=-2 ,14%-3=2

Real Arithmetic:

An arithmetic operation involving only real operands is called real arithmetic. A real value can assume value either in decimal or exponential notation. % operator cannot be used with realoperands
E.g. If a=2.4 and b=1.2 then
a+b=3.6 ,

a-b=1.2,
a*b=2.88a/b=2.0

Mixed mode Arithmetic:

When one of the operands is real and the other is an integer, the expression is called mixed mode arithmetic expression. The result is always a real number
E.g. : 15/10.0=1.5 ,
15/10=1.5

Relational Operators:

Comparisons can be done using relational operators. A relational expression contains a combination of arithmetic expressions, variables or constants along with relational operators. A relational expression can contain only two values i.e. true or false. When the expression is evaluated as true then the compiler assigns a non zero(1) value and 0 otherwise. These expressions are used in decision statements to decide the course of action of a running program.
Syntax: ae1 relational operator ae2
where ae1 and ae2 are arithmetic expressions

Operators	Meaning
<	Less than
<=	Less than or equal to
>	Greater than
y>=	Greater than or equal to
==	Equal to
!=	Not equal to

Example:4.5<= 10 true
a+b == c+d
10< 7+5
-35 >= 0

Logical Operators:

An expression that combines two or more relational expressions is called a logical expression and the operators used to combine them are called logical operators.
Syntax: R1 operator R2
where R1 and R2 are relational expressions

Operator	Meaning
&&	Logical And
||	Logical Or
!	Logical Not

Example :
if(age > 55 && salary < 1000)
if(number < 0 || number > 100)

Relational	Relational	R1&&R2	R1||R2

Expression R1	Expression R2		
0	0	0	0
0	Non zero	0	Non zero
Non zero	0	0	Non zero
Non zero	Non zero	Non zero	Non zero

Assignment Operators:

Assignment operators are used to assign the result of an expression to variable. The operator used for assignment in C is '='. C also supports short hand assignment operators like

v op = expression where v- variable and op- operator. Here op= is known as a short had assignment operator.

v op=expression is equivalent to **v=v op expression**

E.g. The statement x+=y+1; is equivalent to x=x+y+1;

<u>Statement with simple Assignment operator</u>

a=a+1	**a+=a**
a=a-1	**a-=a**
a=a*(n+1)	**a*=(n+1)**

The advantage of short hand assignment operator is

a) Easy to read and write
b) Statements are more concise and efficient

Increment/Decrement Operators:

C has 2 very powerful operators that are not found in any other language. They are increment/decrement operators i.e. ++ and --. The ++ operator is used to increment value of a variable by 1. The -- operator is used to decrement value of a variable by 1.

Both are unary operators and can be written as follows: ++m, m++ , m—and –m. Both m++ and ++m mean the same thing when they form statements independently. But, they behave differently when used in expression on the right hand side of assignment operator.

E.g. Let a=5; x=a++; Here, this statement can be broken into 3 statements as follows a=5; x=a; a=a+1;

In the above example, the value of a is assigned to x and then its value is incremented by 1**. A prefix operator first adds 1 to the operand and then the result is assigned to the variable on the left. A postfix operator first assigns its value to the variable and then increments its value by 1.**

E.g. m=10;
y=--m

Here the value of m is decremented by 1 and then assigned to y. Hence the value of y is 9.

Example:Y= ++m In this case value of y and m would be 6
Y= m++ In this case vale of y would be 5 and m would be 6.

Example:
```
#include<stdio.h>
#include<conio.h>
void main()
    {
```

```
intn,k,m;
clrscr();
      k=1,m=2;
       n= k++ + m;
printf("%d",k);
printf("%d",n);
getch();

}
Output:k=2, n=3
```

Difference i++ and ++i with example:

They are unary operators

- When postfix++(or--)is used with a variable in an expression ,the expression is evaluated using the original value of the variable and then the variable is increment.ex m=a++; m=a--
- When prefix++(or--)is used in a expression, the variable is incremented and then the expression is evaluated using the new value of the variable.ex:m=++a m=--a

Conditional Operator:

Conditional operators are also called ternary operators. Conditional expressions can be constructed in C using the operator pair '?:'.

Syntax: expr?expr1:expr2;

Here expr is evaluated first. If the value is true, then expr1 is evaluated and becomes the value of the expression. If the expression is false then, expr2 is evaluated and becomes the value of the expression.

E.g. a=5;b=10;
    x=(a>b)?a:b;

The output of the above example is as follows- The value of b i.e. 10 is assigned to x.

Bitwise Operators:

These operators are used to manipulate data at bit level. These operators are used for testing the bits or shifting the bits either to the left or right.

Operator	Meaning
&	Bitwise And
|	Bitwise Or
^	Bitwise exclusive Or
<<	Shift left
>>	Shift Right
~	One's complement

Special Operators:

C supports other special operators like pointer operator (&,*) and member selection operator(.). Comma is also an operator use to link related expressions together. The sizeof operator is a compile time operator. It returns the number of bytes occupied by the operand depending on the data type. The operand could be a variable, constant or a data type qualifier.

Comma Operator : Comma Operator can be used to link the related expression together.expressions are evaluated *to left to right* and the value *of right-most* expression is the value of the combined expression.

Ex:Value = (x=10, y=5, x+y); value=15

```
#include<stdio.h>
void main()
{
inta,b,c;
c=(a=15,b=10);
printf("The value of c is = %d",c);
getch();
}
Output
The value of c is = 10
```

Sizeof operator is a compile time operator. It returns number of bytes occupied by the operand. The operand may be a variable, constant or a data type.

```
Ex:m=sizeof(sum);
n=sizeof(long int)
```

```
Example: #include<stdio.h>
void main()
{
float x;
printf("The size of variable x is %d bytes",sizeof(x));
getch();
}
Output
The size of variable x is 4 bytes
```

Arithmetic Expressions:

An arithmetic expression is a combination of variables, constants and operators arranged as per the syntax of the language. Expressions are evaluated using assignment statements of the form

variable=expression;

Some of the examples of C expressions are shown below.

Alygebraic expression	**C expression**
• ab-c	a*b-c
• (m+n) (x+y)	(m+n)*(x+y)
• Ab/c	a*b/c
• $3x^2 + 2x+1$	3 * x * x +2 * x+ 1

Evaluation of expression and precedence of arithmetic operators with examples

Expressions are always evaluated from left to right, using the rules of precedence of operators, if parenthesis are missing

High Precedence- * / %

Low Precedence - + -

Basically the evaluation procedure involves 2 left to right passes. During the first pass the high priority operators are applied as they are encountered and during the second pass, lower priority operators are applied as they are encountered.

Example: Consider the following expression

x= a-b/3+c*2-1 If the values of a=9, b=12,c=3

Now x= 9-$\underline{12/3}$+$\underline{3*2}$-1

Pass 1: x=9-4+6-1

Pass 2: x=5+6-1=10

Example 2:9-12/(3+3)(2-1)*

Whenever parentheses are used, the expressions within parentheses assume highest priority. If two or more sets of parentheses appear one after another as shown above, the expression contained in the left-most set is evaluated first and the right-most in the last. Given below are the new steps.

First pass

Step 1: 9-12/6 * (2-1)

Step 2 : 9-12/6 * 1

Second pass

Step 3 : 9-2* 1

Step 4 : 9-2

Third pass

Step 5 :7

Precedence of operators:

If more than one operators are involved in an expression then, C language has predefined rule of priority of operators. This rule of priority of operators is called operator precedence.

In C, precedence of arithmetic operators(*,%,/,+,-) is higher than relational operators(==,!=,>,<,>=,<=) and precedence of relational operator is higher than logical operators(&&, || and !).

Suppose an expression:

(a>b+c&&d)

This expression is equivalent to ((a>(b+c))&&d)

i.e. (b+c) executes first then,

(a>(b+c)) executesthen, (a> (b=c)) && d) executes

Associativity of operators

Associativity indicates in which order two operators of same precedence(priority)

executes. Let us suppose an expression:
a==b!=c
Here, operators == and != have same precedence. The associativity of both == and != is left to right, i.e, the expression in left is executed first and execution take place towards right. Thus, a==b!=c equivalent to :(a==b)!=c

The table below shows all the operators in C with precedence and associativity.

Note: Precedence of operators decreases from top to bottom in the given table.

Summary of C operators with precedence and associativity

Operator	**Meaning of operator**	**Associativity**
() [] -> .	Functional call Array element reference Indirect member selection Direct member selection	Left to right
! ~ + - ++ -- & * Sizeof (type)	Logical negation Bitwise(1 's) complement Unary plus Unary minus Increment Decrement Dereference Operator(Address) Pointer reference Returns the size of an object Type cast(conversion)	Right to left
* / %	Multiply Divide Remainder	Left to right
+ -	Binary plus(Addition) Binary minus(subtraction)	Left to right
<< >>	Left shift Right shift	Left to right
< <= > >=	Less than Less than or equal Greater than Greater than or equal	Left to right
== !=	Equal to Not equal to	Left to right
&	Bitwise AND	Left to right
^	Bitwise exclusive OR	Left to right
|	Bitwise OR	Left to right
&&	Logical AND	Left to right
||	Logical OR	Left to right
?:	Conditional Operator	Left to right
=	Simple assignment	Right to left

Rules for evaluation of expressions

1. First parenthesized sub expression from left to right are evaluated.

2. If parentheses are nested, the evaluation begins with the innermost sub expression
3. The precedence rule is applied in determining the order of application of operators in evaluating sub expressions
4. The associatively rule is applied when 2 or more operators of the same precedence level appear in a sub expression.
5. Arithmetic expressions are evaluated from left to right using the rules of precedence
6. When parentheses are used, the expressions within parentheses assume highest priority

Evaluate x=--c/2+b*8/b-b+a++/3

When a=4,b=3,c=25

X=--25/2+3*8/3-3+4++ /3

=--25/2+3*8/3-3+4/3

=24/2+3*8/3-3+4/3

=12+3*8/3-3+4/3

=12+24/3-3+4/3

=12+8-3+4/3

=12+8-3+1

=20-3+1

=17+1=18

Type conversion :

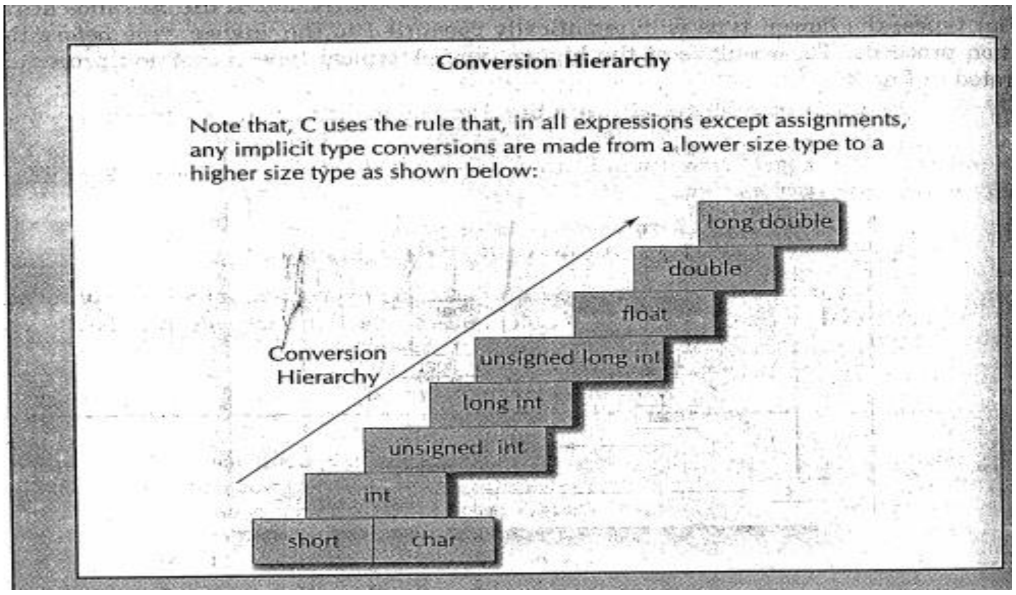
Type casting is a way to convert a variable from one data **type** to another data **type**.

There are two types of the type conversions:

- Implicit Type Conversion
- Explicit Type Conversion

Implicit Type Conversion :

- When data value automatically convert from one type to another type is called the Implicit type conversion.
- If the operands are of different types, the lower type is automatically converted to the higher type before the operation proceeds.


Conversion Hierarchy
Note that, C uses the rule that, in all expressions except assignments, any implicit type conversions are made from a lower size type to a higher size type as shown below:
Conversion Hierarchy
long double
double
float
unsigned long int
long int
unsigned int
int
short
char

1.float to int causes truncation of the fractional part.
2.double to float cause rouding of digits.
3.longint to int cause dropping of the excess higher order bits.

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a,b;
float c;
a= 12,b=13;
c= a+b;    //automatically conversion   done
printf("%f",c);
getch();
}
```

Explicit Type Conversion:

When a process of the conversion done manually of locally then this process is called the explicit type conversion process or casting operation.

Syntax :(type_name) expression

Ex:

Example	*Action*
x =(int)7.5	7.5 is converted to integer bytruncation.
a =(int)21.3/(int)4.5	Evaluated as 21/4 and the result would be5.
b=(double)sum/n	Division is done in floating pointmode.

y =(int)(a+b)	The result of a+b is converted tointeger
z=(int)a+b	a is converted to integer and then added tob.

example:

```
#include<stdio.h>
 #include<conio.h>
void main()
{
floata,b;
int c;
clrscr();
 a=12.3;
b=13.3;
c= (int) (a+b);
printf("%f",c);
getch();
}
/* Output 25 */
```

Note : whenever you try to convert higher data type to lower type your result will be loss.

Built-in Mathematical functions:

Mathematical functions such as cos, tan and sqrt are widely used in problem solving. All these functions are available in the <math.h> header file. It is necessary to include this header file in the program in order to use these functions. The list of mathematical functions available is given below:

Function	Meaning
ceil(x)	x rounded up to the nearest integer
exp(x)	e to the power of x
fabs(x)	Absolute value of x
floor(x)	x rounded down to the nearest integer
log(x)	Natural log of x, x>0
pow(x,y)	x raised to y (x^y)
sqrt(x)	Square root of x ; x>=0

Unit 2
Chapter 2
Input and Output Statements in C

Input means to provide the program with some data to be used in the program and **Output** means to display data on screen or write the data to a printer or a file.
C programming language provides many built-in functions to read any given input and to display data on screen when there is a need to output the result.
All these built-in functions are present in C header files;

The getchar() and putchar() Functions [Unformatted I/O]

The **getchar()** function reads the character from the keyboard. This function reads only single character at a time. To continuously read the characters use getchar() within looping statement.

Syntax : c=getchar() where c is character variable.

The **putchar(int c)** function printsthe given character on the screen . This function puts only single character at a time. You can use this method in the loop in case you want to display more than one character.

```
#include <stdio.h>
main( )
{
   int c;
printf( "Enter a value :");
   c = getchar( );
printf( "\nYou entered: ");
putchar(c);
}
```

When the above code is compiled and executed, it waits for you to input some text. When you enter a text and press enter, then the program proceeds and reads only a single character and displays it

Formatted input/output - scanf() and printf() functions

The standard input-output header file, named stdio.h contains the definition of the functions printf() and scanf()

scanf() function

Syntax :

scanf("format string", arg1,arg2…argn);

- This function is usually used as an input statement.
- The format string must be a text enclosed in double quotes. It contains type of data to input. Example: integer (%d) , float (%f) , character (%c) or string (%s).

- The arg1,arg2..argn contains a list of variables each preceded by & (to get address of variable) and separated by comma.

Commonly used format specifiers are as follows

%c character	%f floating point
%d integer	%lf double floating point
%i integer	%e exponential notation
%u unsigned integer	%s string
%ld signed long	%x hexadecimal
%lu unsigned long	%o octal

Examples :-

```
int i, d ;
char c ;
float f ;
scanf( "%d", &i ) ;  /* input integer data*/
scanf( "%d %c %f", &d, &c, &f ) ; /* input int , char and float */
```

The & character is the *address of* operandin C, it returns the address in memory of the variable it acts on.

printf() function

Syntax **:**
printf("format string", v1,v2…vn);

The printf() function is used for formatted output and uses a format string which is made up of a series of format specifiers to govern how it prints out the values of the variables or constants required. The more common format specifiers are given below

%c character	%f floating point
%d integer	%lf double floating point
%i integer	%e exponential notation
%u unsigned integer	%s string
%ld signed long	%x hexadecimal
%lu unsigned long	%o octal

```
#include<stdio.h>
void main()
{
  int i;
printf("Please enter a value...");
scanf("%d", &i);
printf( "\nYou entered: %d", i);
}
```

When you will compile the above code, it will ask you to enter a value. When you will enter the value, it will display the value you have entered on screen.

Some further examples :-

```
int i = 10, j = 20 ;
char ch = 'a' ;
double f = 23421.2345 ;
printf( "%d + %d", i, j ) ;  /* values are substituted from the variable list  in order as required
*/
printf( "%c", ch ) ;
printf( "%s", "Hello World\n" ) ;
printf( "The value of f is : %lf", f ) ;          /*Output as : 23421.2345 */
printf( "f in exponential form : %e", f ) ;          /* Output as : 2.34212345e+4*/
```

Field Width Specifiers

Field width specifiers are used in the control string to format the numbers or characters output appropriately .

***Syntax :-* %[total width][.decimal places]format specifiers**

where square braces indicate optional arguments.

For Example:-

```
int i = 15 ;
float f = 13.3576 ;
printf( "%3d", i ) ;   /* prints "_15 " where _ indicates a space  character */
printf( "%6.2f", f ) ; /* prints "_13.36" which has a total width of 6 and displays 2 decimal places  */
```

gets() & puts() functions

gets() functions reads a line of string and store it in a string variable.
Syntax : gets(str) where str is string variable

puts() functions prints string .
Syntax :puts(str) where str is string variable

Example:

```
#include<stdio.h>
void main()
{
   /* character array of length 100 */
   char str[100];
printf("Enter a string");
gets( str );
puts( str );
getch();
```

}

Difference between scanf() and gets()

The main difference between these two functions is that scanf() stops reading characters when it encounters a space, but gets() reads space as character too.

If you enter name as **Hello World** using scanf() it will only read and store **Hello** and will leave the part after space. But gets() function will read it completely.

UNIT2
CHAPTER 3
Conditional and Loop Control Structures

C conditional statements allow you to make a decision, based upon the result of a condition. These statements are called as **Decision Making Statements** or **Conditional Statements**.

If statement(s)

If statements in C is used to control the program flow based on some condition, it's used to execute some statement code block if the expression is evaluated to true. Otherwise, it will get skipped. This is the simplest way to modify the control flow of the program.

There are four different types of if statement in C. These are:

- Simple if Statement
- if-else Statement
- Nested if-else Statement
- else-if Ladder

Simple if Statement

The basic format of if statement is:

```
if(test_expression)
{
   statement 1;
   statement 2;
...
}
```

Here if the test expression is evaluated to true, the statement block will get executed, or it will get skipped.

Example :

```
#include<stdio.h>

main()
{
  int a = 15, b = 20;

  if (b > a) {
printf("b is greater");
  }
}
```

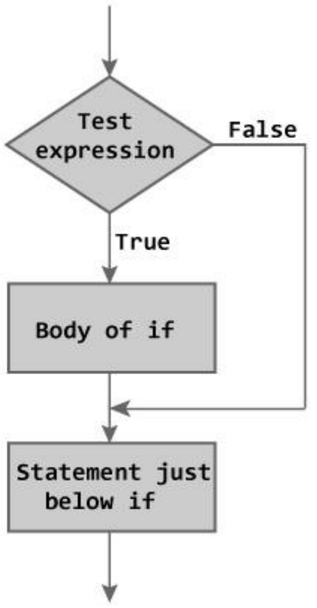


Figure: Flowchart of if Statement

If ..else Statement

The syntax of an **if...else** statement in C programming language is −

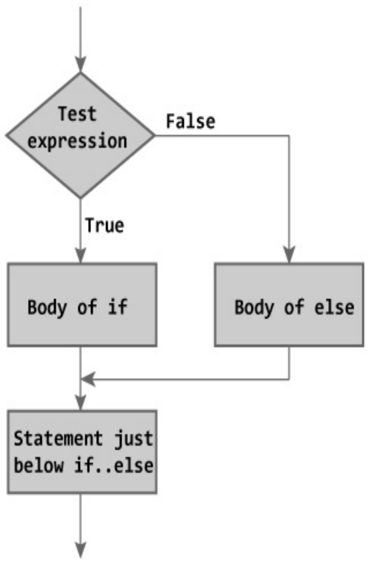


Figure: Flowchart of if...else Statement

```
if(test expression)
{
  Statement block-1
}
 else
{
  Statement block-2
}
```

If the test expression evaluates to true, then the if block will be executed, otherwise, the else block will be executed.

Example:

```
#include<stdio.h>

main( )
   {
    int a;
printf("n Enter a number:");
scanf("%d", &a);
    if(a>0)
printf( "n The number %d is positive.",a);
    else
printf("n The number %d is negative.",a);
  }
```

nested-if Statement

A nested if is an if statement that is inside another if statement .

Syntax:

```
if (condition1)
{
/* Executes when condition1 is true*/
   if (condition2)
   {
/* Executes when condition2 is true*/
   }
}
```

Example

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

```
{
  int var1, var2;
printf("Input the value of var1:");
scanf("%d", &var1);
printf("Input the value of var2:");
scanf("%d",&var2);
  if (var1 != var2)
  {
          printf("var1 is not equal to var2\n");
          /*Nested if else*/
          if (var1 > var2)
          {
                  printf("var1 is greater than var2\n");
          }
          else
          {
                  printf("var2 is greater than var1\n");
          }
  }
  else
  {
          printf("var1 is equal to var2\n");
  }
  return 0;
}
```

if-else-if ladder

Here, a user can decide among multiple options. The if statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final else statement will be executed.

Syntax:

```
if (condition-1)
{
   Statement block-1;
}
else if (condition-2)
{
   Statement block-2;
}
.
.
Else
{
Default block of  statement(s);
}
```

The if else ladder statement in C is used to test set of conditions in sequence. An if condition is tested only when all previous if conditions in if-else ladder is false. If any of the conditional expression evaluates to true, then it will execute the corresponding code block and exits whole if-else ladder.

- First of all condition1 is tested and if it is true then statement block-1 will be executed and control comes out of whole if else ladder.

- If condition1 is false then only condition2 is tested. Control will keep on flowing downward, If none of the conditional expression is true.

- The last else is the default block of code which will gets executed if none of the conditional expression is true.

Example

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

int main(){
    int marks;

    printf("Enter your marks between 0-100\n");
    scanf("%d", &marks);
    if(marks >= 90){
        printf("YOUR GRADE : A\n");
    else if (marks >= 70 && marks < 90)
        printf("YOUR GRADE : B\n");
    else if (marks >= 50 && marks <70){
        printf("YOUR GRADE : C\n");
     else
        printf("YOUR GRADE : Failed\n");

}
```

Switch Statement

Syntax :

```
switch (variable or integer expression)
{
    case constant1:
     Statement(s) ;
break;

    case constant2:
            Statement(s) ;
 break;
        .
        .
        .
    default:
```

default statement(s);
}

Switch statement is a control statement that allows us to choose only one choice among the many given choices. The expression in switch evaluates to return an integral value, which is then compared to the values present in different cases. It executes that block of code which matches the case value. If there is no match, then **default** block is executed(if present).

Rules for using switch statement

1. The expression (after switch keyword) must yield an **integer** value i.e the expression should be an integer or a variable or an expression that evaluates to an integer.
2. The case **label** values must be unique.
3. The case label must end with a colon(:)
4. The next line, after the **case** statement, can be any valid C statement.

Difference between switch and if

- if statements can evaluate float conditions. switch statements cannot evaluate float conditions.
- if statement can evaluate relational operators. switch statement cannot evaluate relational operators i.e they are not allowed in switch statement.

Example :

```
#include <stdio.h>
main()
{
  int x = 2;
  switch (x)
  {
      case 1:
printf("Choice is 1");
break;
      case 2:
printf("Choice is 2");
 break;
      case 3: printf("Choice is 3");
             break;
      default: printf("Choice other than 1, 2 and 3");
              break;
  }
  return 0;
}
```

Goto Statement

The goto statement is a jump statement which is sometimes also referred to as unconditional jump statement. The goto statement can be used to jump to specified label within a function.

Syntax:

goto label;

….

….

label: statement(s);

example:

```
#include<stdio.h>
main()
{
int sum=0;
for(int i = 0; i<=10; i++){
            sum = sum+i;
            if(i==5){
            goto addition;
            }
  }

  addition:
printf("%d", sum);

}
```

UNIT 3
CHAPTER 1
Decision making and looping

You may encounter situations, when a block of code needs to be executed several number of times. A loop statement allows us to execute a statement or group of statements multiple times. Repeated execution of statement or statements is called looping.
C programming language provides the following types of loops

Advantage with looping statement

- Reduce length of Code
- Take less memory space.
- Burden on the developer is reducing.
- Time consuming process to execute the program is reduced.

There are three type of Loops available in 'C' programming language.

- while loop
- do..while
- for loop

There are mainly two types of loops:

1. **Entry Controlled loops**: In this type of loops the test condition is tested before entering the loop body. **For Loop** and **While Loop** are entry controlled loops.
2. **Exit Controlled Loops**: In this type of loops the test condition is tested or evaluated at the end of loop body. Therefore, the loop body will execute atleast once, irrespective of whether the test condition is true or false. **do – while loop** is exit controlled loop.

While Loop

The syntax of a while loop is:

```
while (testExpression)
{
  Statement block
}
```

where, test Expression checks the condition is true or false before each loop.

The while loop evaluates the test expression. If the test expression is true (nonzero), codes inside the body of while loop are executed. The test expression is evaluated again. The process goes on until the test expression is false.When the test expression is false, the while loop is terminated. While loop is entry controlled loop.

Example

/* Programm to print numbers from 1 to 9*/:

```
#include <stdio.h>
main()
{
  int i=1;

  while(i<10)
  {
     printf("%d\n",i);
     i++;
  }

}
```

Do ..while loop

do...while loop in C programming checks its condition at the bottom of the loop.

A do...while loop is similar to a while loop, except the fact that it is guaranteed to execute at least one time.

Syntax :

```
do {
   statement(s);

} while(condition );
```

if the condition is true, the flow of control jumps back up to do, and the statement(s) in the loop executes again. This process repeats until the given condition becomes false.

Example

```
 main ()
{
    int a = 10;
   /* do loop execution */
   do {
printf("  %d\n", a);
      a = a + 1;
}while( a < 20 );
 }
```

When the above code is compiled and executed, it prints values from 10 to 19

for loop

A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

Syntax

The syntax of a **for** loop in C programming language is −

```
for ( init_exp; test_exp; update_exp )
{
  statement(s);
}
```

- The **init_exp** step is executed first, and only once. This step allows you to initialize loop control variables.
- Next, the **test_exp** is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and the flow of control jumps to the next statement just after the 'for' loop.
- After the body of the 'for' loop executes, the flow of control jumps back up to the **update_exp** This statement allows you to update any loop control variable
- The condition is now evaluated again. If it is true, the loop executes and the process repeats itself. After the condition becomes false, the 'for' loop terminates.

Example

```
#include <stdio.h>
main ()
 {
int  i;
   /* for loop execution */
for( i = 1;i < 10;i++ ){
printf("%d\n",,i);
   }
}
```

When the above code is compiled and executed, it prints values from 1 to 9

Omitting expressions in for loop

All the three expression inside the for loop is optional. You are free to omit one or all three expressions, but in any case, two semicolons must be present.

- init_exp can be omitted when initialization is done outside the for loop.
- test_expis omitted then the condition is always assumed to be true and so this type of loop never stops executing. This type of loop is known as an infinite loop. To avoid thisyou should include a statement that takes you out of the for loop.
- update_exp is omitted when update statement is present inside the body of the for loop.

example 1:

```
#include<stdio.h>

main()
{
   int i = 1, sum = 0;
   /*expression 1 is omitted*/
for( ; i < 10 ; i++)
```

```
    {
        sum += i;
    }
printf("Sum = %d", sum);
}
```

Example 2:

2nd variation: Where expression2 is omitted

```
#include<stdio.h>
main()
{
    int i, sum = 0;

for(i = 1 ; ; i++)
   {
if(i > 1000)
       {
           break;
        }
        sum += i;
    }
printf("Sum = %d", sum);
}
```

Example 3:

3rd variation: Where expression3 is omitted

```
#include<stdio.h>
main()
{
    int i, sum = 0;
for(i = 1 ; i < 10; )
    {
        sum += i;
        i++; /* update expression*/
    }
printf("Sum = %d", sum);

}
```

Nesting of Loops

C programming allows to use one loop inside another loop. The following section shows a few examples to illustrate the concept.

The syntax for a **nested for loop** statement in C is as follows −

```
for ( init_exp; test_exp; update_exp )
 {

for(init_exp; test_exp; update_exp)
{
    statement(s);
  }
  statement(s);
}
```

The syntax for a **nested while loop** statement in C is as follows −

```
while(condition) {

  while(condition) {
    statement(s);
  }
  statement(s);
}
```

The syntax for a **nested do...while loop** statement in C is as follows −

```
do {
  statement(s);

  do {
    statement(s);
}while( condition );

}while( condition );
```

Example

C program to print the number pattern.

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

```
#include <stdio.h>
main()
```

```
{
   int i=1,j;
   while (i <= 5)
   {
      j=1;
      while (j <= i )
      {
printf("%d ",j);
         j++;
      }
printf("\n");
      i++;
   }
   }
```

The Infinite Loop

A loop becomes an infinite loop if a condition never becomes false. Loop will not terminate

Example

```
int i = 1
while(i<10)
{
printf("%d\n", i);
}
```

Here we are not updating the value of i. So after each iteration value of i remains same. As a result, the condition (i<10) will always be true. For the loop to work correctly add i++;

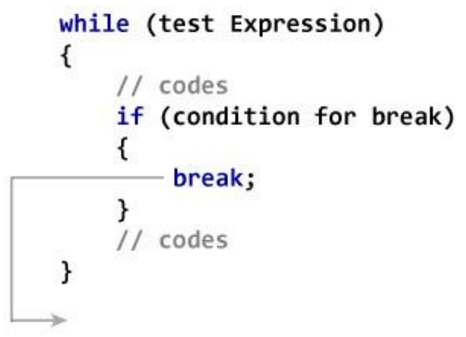
break Statement

The break statement terminates the loop (for, while and do...while loop) immediately when it is encountered. The break statement is used with decision making statement such as if...else.

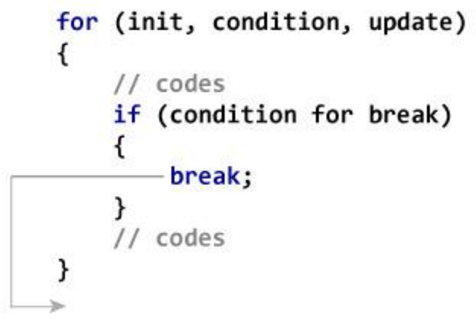
Syntax of break statement

```
break;
```

How break statement works?



```
while (test Expression)
{
    // codes
    if (condition for break)
    {
        break;
    }
    // codes
}
```



```
for (init, condition, update)
{
    // codes
    if (condition for break)
    {
        break;
    }
    // codes
}
```

Example

```
 main ()
{

   int a = 1 ;
  /* while loop execution */
while( a< 10 )
  {
printf("value of a: %d\n", a);
     a++;
if( a>  5)
      {

        /* terminate the loop using break statement */
        break;
     }
  }


}
```

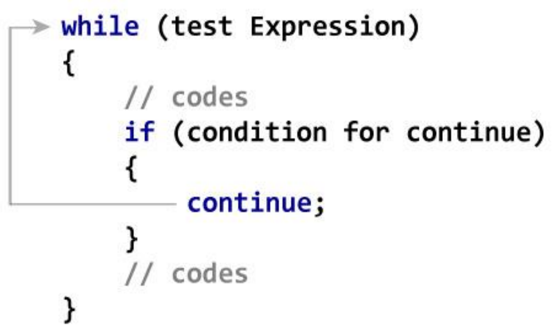
continue Statement

The continue statement skips some statements inside the loop and goes back to beginning of loop for executing next iteration.. The continue statement is used with decision making statement such as if...else.

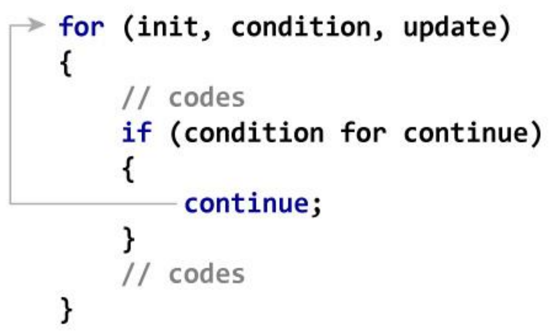
Syntax of continue Statement

continue;

How continue statement works?



```
while (test Expression)
{
    // codes
    if (condition for continue)
    {
        continue;
    }
    // codes
}
```



```
for (init, condition, update)
{
    // codes
    if (condition for continue)
    {
        continue;
    }
    // codes
}
```

Example

```
#include <stdio.h>
  main ()
{
  int a = 1 ;
   /* do loop execution */
   do {
if( a ==  5)
{
        a ++;
        continue;
     }
printf("value of a: %d\n", a);
      a++;
      } while( a< 10 );
}
```

UNIT 3
CHAPTER 2
ARRAYS

Introduction
So far we have used only the fundamental data types, namely **char, int, float, double** and variations of **int**and **double**. Although these types are very useful, they are constrained by the fact that a variable of these types can store only one value at any given time. Therefore, they can be used only to handle limited amounts of data. In many applications, however, we need to handle a large volume of data in terms of reading, processing and printing. To process such large amounts of data, we need powerful data type that would facilitate efficient storing, accessing and manipulation of data items. C supports a derived data type known as ***array*** that can be used for such applications.

Important Definition:
 “Array is a group of the related data items that share a common name.”
 “Individual elements of array are known as the elements of array.”
 “The number specified within the square bracket during the declaration of variable is known as the array size.”
e.g. int a[20];

Here the 20 is the size of the array since it is specified during the declaration of array.
 “The number given to each elements is known as the index or subscript.”
In ‘C’ program the array index start from 0 and goes up to the (size of array –1)
e.g.
a[0]
a[1]
a[2]
…….
Here the number given from 0, 1, 2, … are known as the index

<u>One-dimensional array (or) single-subscripted array:</u>
One-dimensional array store the list of values. “An array with only one subscript is known as the one-dimensional array of single-subscripted variable.”
e.g. int a[5];
Here we have the array with the size of 5 that means it can store maximum 5 integer elements that means we can store 5 different integer values inside it. The subscript (index) or array is start with the 0 and up to 4 (size –1)

If you want to read of print array element then you can do by:
e.g. scanf(“%d”,&x[0]); // read first element
scanf(“%d”,&x[1]); // read second element
…….
printf(“%d”,x[0]); // print first element
printf(“%d”,x[1]); // print second element
……..

You can read/print the five elements by using only one scanf/printf statement as under with loop:

```
e.g. for(i=0 ; i<5 ; i++)
 {
scanf("%d",&x[i]); // read 1 to 5 element as loop will increment
printf("%d",x[i]); // print 1 to 5 element as loop will increment
}
```

You can perform the calculation also by using the subscript:
e.g total = x[0] + x[1] + x[2] + x[3] + x[4];
This will determine the total of five elements:

If suppose we assign the values 5, 10, 15, 20, and 25 to each array elements respectively:

```
X[0] = 5;
X[1] = 10;
X[2] = 15;
X[3] = 20;
X[4] = 25;
```

Then each value is successively stored into the array as under inside memory:

<u>Declaration of Array:</u>

Like any other variables, arrays must be declared before they are used. The general format of declaration of array is:

```
datatypearrayname[size];
```

The 'datatype' indicates whether your array is integer type, float type or any other type. 'arrayname' is a name given to array, and 'size' indicates the no. of elements in your array.

e.g. float height[10];

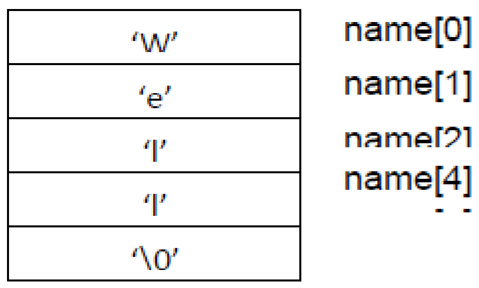
It means out array name is 'height' of float type and it contains different 10 elements. Here the valid subscript range is 0-9.

q)Explain the representation of String using array of the character:

In 'C' program the array of character represents the string. In string the size indicates your string contains how many maximum character.

e.g. char name[5];

Here we have the array of the character 'name' which is known as string and here its size is 5 and you can store maximum 4 characters in your string because in string the last character is null character '\0' or NULL. Suppose we are storing string "Well" in array 'name' then it is stored as:



As shown above all the characters of the string are stored in continuous memory location and the last character of your string is NULL character.

ACCESSING ARRAY ELEMENTS

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array.
For example: int salary = balance[9];
The above statement will take 10th element from the array and assign the value to salary variable

Initialization of One-dimensional Array:
The general format of the initialization of array is:

- **At compile time**
- **At run time**

CompileTime Initialization:

Syntax:
datatypearrayname[size]={list of values separated by comma};

To initialize your array you have to give list of the values separated by comma in curly bracket.
e.g. int x[5] = { 10, 20, 30, 40, 50};
The array variable of size 5 ‘x’ is initialized by five different values 10, 20, 30, 40 and 50 respectively that means
X[0] = 10;
X[1] = 20;
X[2] = 30;
X[3] = 40;
X[4] = 50;

You can initialize the character array or string by two ways:

e.g. char name[5] = { ‘W’,’e’,’l’,’l’,’\0’};
or
char name[5] = “Well”;
You can initialize the array elements without specifying the size of array
e.g. int x[ ] = { 10, 20, 30, 40, 50};
Here we have given the empty bracket and initialized by five variables that means by default the size is 5.
Initialization of array has two drawbacks or limitation:
 There is no any way to initialize selected array elements.
 There is no any shortcut method to initialize the large no. of array elements.

(Note:
1. In old version of ‘C’ we can initialize the array variable of only static or extern type but not auto type variable but in ANSI version of ‘C’ we can initialize any array elements.

2. If you initialize the less no. of elements then the given size then other elements are by default initialized by zero.

e.g. int x[5] = { 10, 50};
Here the sized of array is 5 but we have initialized only 2 elements so only first two elements are initialized other are initialized by zero.)

PROGRAM – 2 initialize some values to an array at time of compilation and find the total and average of those number.

```
main()
{
int a[5]={20,30,40,50,60};
inti,sum=0;
float avg;
for(i=0;i<=4;i++)
{
sum=sum+a[i];
}
avg=sum/5;
printf("The sum is %d", sum);
printf("The average is %f", avg);
}
```

<u>Run Time Initialization :</u>
An array can be explicitly initialized at run time. This approach is usually applied for initializing large arrays.
Ex:

```
for(i=0; i<100 ; i++)
{
if (i < 50 )
sum[i] = 0.0;
else
sum[i] = 1.0;
}
```

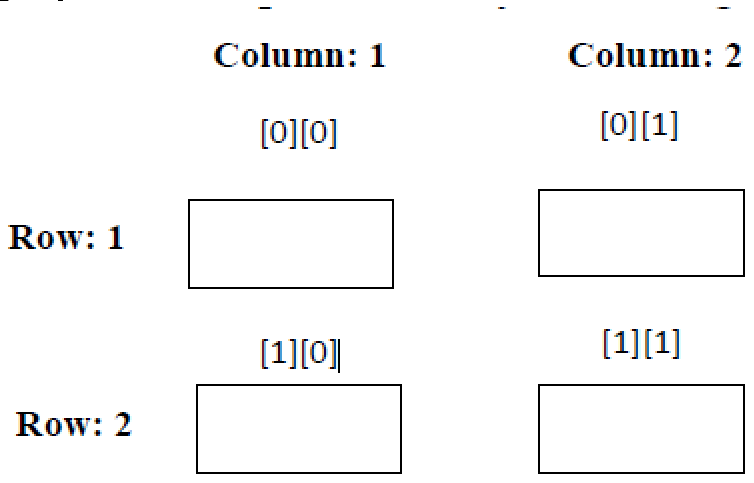
In the above case the first 50 elements are initialized to zero while the remaining elements are initialized to 1.0 are run time.

Two-dimensional Array:
Two-dimensional array can store the table of values. "An array with two subscript is known as the two-dimensional array of double-subscripted variable."
The two-dimensional array can be represented by the matrix in mathematics. The matrix elements can be represented by (i j) where 'i' value indicates the row number and 'j' value indicated the column number.

We can declare the two-dimensional array by using following format:
datatypearrayname[row size][column size*];*

e.g. int x[2][2];

Means your array ‘x’ contains 2 rows and 2 columns. The array elements can be represented by following way:



e.g. Suppose we have to read the value of the two-dimensional array and print then:

```
main( )
 {
int a[5][5]; // Declaration of array with the size = 5
int i, j;
// read the array elements value
for(i = 0 ; i < 5 ; i ++)
{
for(j = 0 ; j < 5 ; j++){
scanf(“%d”,&a[i][j]);
}
}
// print the array elements value
for(i = 0 ; i < 5 ; i ++)
 {
for(j = 0 ; j < 5 ; j++)
{
printf(“%d”,a[i][j]);
}
}
```

Initialization of Two-dimensional Array:

We can initialize two-dimensional array elements by following way:
datatypearrayname[row size][column size] = { {list of values of 1st row},
{list of values of 2nd row},
…… };

We can also initialize the array elements continuously as:

datatypearrayname[row size][column size] = { {list of values };

Here the elements are initialized continuously from first row, then second row, and so on.
e.g:inta[2][2] = {{0,0}, {1,1}};

OR
int a[2][2] = {0,0,1,1};
Here the first row elements are initialized by zero and second rows elements are initialized by 1.
Example:

```
#include<stdio.h>
 #include<conio.h>
void main()
{
inti,j,n,m,a[10][10],b[10][10];
printf("Enter No of Rows for first matrix:");
scanf("%d",&n);

printf("Enter No of columns for first Matrix:");
scanf("%d",&m);
for(i=0;i<n;i++)
{
for(j=0;j<m;j++)
{
scanf("%d",&a[i][j]);
}
}
printf("After Sum New Matrix\n");
for(i=0;i<n1;i++)
{
for(j=0;j<m1;j++)
{
printf("%d ",a[i][j]+b[i][j]);
}
printf("\n");
}
}
getch();
}
```

Multidimensional array:
In multidimensional array we have more than two subscripts. Its format is:

datatypearrayname[s1][s2][s3][s4][s5]….[sn];

Here the ‘si’ indicates the size of the ‘i’th dimension.
e.gint marks[2][80][6];
Here ‘marks’ is the array of 3-dimension. Here first subscript indicates the no. of division, 2nd subscript indicates the no. of students, and the 3rd subscript indicates the no. of subjects. In our case there are two division, 80 students, and 6 subjects.
q)write program transpose matrix
Program to find transpose of the matrix.

```
#include<stdio.h>
void main()
{
```

```
int a[10][10],m,n,i,j,f;
clrscr();
 f=0;
printf("Enter m and n: ");
scanf("%d%d",&m,&n);


printf("Enter the element: ");
for(i=0;i<m;i++)
 {
for(j=0;j<n;j++)
    {
scanf("%d",&a[i][j]);
    }
 }
printf("Entered matrix :\n");
for(i=0;i<m;i++)
  {
for(j=0;j<n;j++)
    {
printf("%d\t",a[i][j]);
    }
printf("\n");
  }
printf("Transpose of the matrix :\n");
for(i=0;i<n;i++)
  {
for(j=0;j<m;j++)
    {
printf("%d\t",a[j][i]);
    }
printf("\n");
  }
Getch();
}
```

q) write program sort element in ascending or descending order.

Program to sort in ascending order using bubble sort.*/

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a[10],temp,n,i,j;
clrscr();
printf("Enter the number of elements: ");
scanf("%d",&n);
printf("Enter the elements: ");
for(i=0;i<n; i++)
 {
scanf("%d",&a[i]);
 }
for(i=1;i<n; i++)
 {
for(j=0;j<n; j++)
  {
if(a[j]>a[j+1])
   {
temp=a[j];
a[j]=a[j+1];
a[j+1]=temp;
   }
  }
 }
printf("\nElements after sorting: ");
for(i=0; i<n; i++)
printf("%d\t",a[i]);
getch();
}
```

Program to find largest and smallest in one dimensional array. Also print their position.

```
#include<stdio.h>
#include<conio.h>
void main()
{
intarr[10],n,i,big,small,pos,pos1;
clrscr();
printf("Enter number of elements: ");
scanf("%d",&n);
printf("Enter the elements: ");
for(i=0;i<n;i++)
 {
scanf("%d",&arr[i]);
 }
big=arr[0];
small=arr[0];
pos=1;
 pos1=1;




for(i=1;i<n;i++)
 {
if(big<arr[i])
    {
```

```
big=arr[i];
pos=i+1;
    }

if(small>arr[i])
    {
small=arr[i];
      pos1=i+1;
    }
  }
printf("The largest number is %d at position %d\n",big,pos);
printf("The smallest number is %d at position %d",small,pos1);
getch();
}
```

UNIT 3
CHAPTER 3

HANDLING OF CHARACTER STRINGS

Introduction

A string is an array of characters. Any group of characters enclosed in double quotes is a string constant.

E.g.: “Hello”

The common operations performed on strings are

1. Reading/Writing
2. Combining strings
3. Copying one string to another
4. Comparing two strings
5. Extracting a portion of strings

<u>Declaring and initializing strings:</u>

A string variable is any valid C variable name and is always declared as an array. The general form of declaration is

```
char string-name[size];
```

where**size** determines the number of characters in the string-name.

When a character string is assigned to a character array, the compiler automatically supplies a null character (‘\0’) at the end of the string. Hence the size should always be equal to the maximum number of characters plus one.

C permits a character array to be initialized in either of the following forms

char city[9]= “NEW YORK”;

char city[9]= {‘N’, ‘E’, ‘W’, ‘ ‘, ‘Y’, ‘O’, ‘R’, ‘K’ ‘\0’};

When we initialize a character array by listing elements, a null character must be specified explicitly.

C also permits to initialize character strings without specifying the number of elements.

char name[]= { ‘G’, ‘O’, ‘O’, ‘D’, ‘\0’};

<u>Reading Strings from Terminals:</u>

Words can be read using the scanf function with %s as the format specifier. But the problem with scanf statement is that it terminates its input when it encounters a white space.

For **example:**

char city[15];

scanf(“%s”, city);

If New York is given as input at the terminal then, only New will be assigned to the variable city.

<u>Reading a Line of Text:</u>

It is not possible to read more than one word because **scanf**terminates as soon as it encounters a white space in the input. Hence**, getchar()** function is used to repeatedly read successive single characters from the terminal and place them into an array. In this case, reading is terminated when a new line character is entered and a null character is appended at the end of the string.

Example:

```
i=0;
while((ch=getchar())!='\n')
  line[i++]=ch;
line[i]='\0';
```

Writing Strings to the Screen:

The format %s can be used to display an array of characters that is terminated by a null character.

For example**: Consider the following statement**

printf("%s", name);

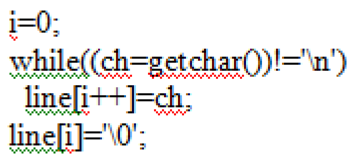
The above statement can be used to display the entire contents of the array, name.

Writing a Line of Text:

putchar() function is used to write a single character on to the screen. This helps us to print a line on the screen. **putchar()** takes one argument i.e. the character that is to be printed on the screen.

Example:

```
Do
{
putchar(ch);
}while(ch!='\n');
```



```
i=0;
while((ch=getchar())!='\n')
  line[i++]=ch;
line[i]='\0';
```

Arithmetic Operations on Strings:

C allows us to manipulate characters the same way we do with numbers. Whenever a character constant or a character variable is used in an expression, it is automatically converted into an integer value by the system. The integer value depends on the local character set of the system.

For example, if the system uses ASCII representation

then, x='a';
printf("%d",x);

The above code will display he number 97 on the screen. 97 is the ASCII equivalent on the letter 'a'.

It is also possible to perform arithmetic calculations on character constants and variables. Consider this statement: x='z'-1;

It is a valid C statement. The ASCII code for z is 122. Hence the above statement will assign the value 121 to the variable x.

Character constants can also be used in relational expressions.

For example: This expression (ch>= 'A' &&ch<= 'Z') would test if the character contained in ch is an upper case letter.

We can also convert character digit to its equivalent integer value using the following statement. x=ch-digit – '0';

Here x is a variable and ch-digit is a character digit.

E.g. Let us assume that ch-digit contains the digit '7', then x= ASCII value of 7 – ASCII value of 0;
= 55-48 = 7

C library also supports a function that converts a string of digits to its equivalent integer value. This is done using the function atoi. The general format of this function is

x= atoi(string);

where x is an integer variable and string is a character array containing a string of digits.

Example: num= "1947";
year=atoi(num);

The atoi function converts the string "1947" to its numeric equivalent 1947 and assigns it to year.

Putting Strings together (Concatenation):

The process of combining 2 strings together is known as concatenation. In C, we cannot directly combine two strings. The characters from string1 and string2 should be copied to string3 one after another. But, the size of string3 should be large enough to accommodate both string1 and string2.

Comparison of two strings:

Strings cannot be compared directly. We have to compare the two strings character by character. The comparison is done till there is a mismatch or one of the strings terminates to a null character, whichever occurs first.

Built-in String Functions:

C library supports a large number of built-in string functions in a header file called string.h. The most important string handling functions are tabulated as follows:

Function	Purpose
strcat()	Concatenates two strings
strcmp()	Compares two strings
strcpy()	Copies one string to another
strlen()	Finds length of a string

<u>strcat() Function:</u>

This function is used to concatenate or join 2 strings together. It takes the following form

strcat(string1,string2);

where string1 and string2 are character arrays. When this function is executed string2 is appended to string1, It does so by removing the null character at the end of string1 and placing string2 from there. The contents of string2 remain unchanged.

E.g.:

word1:

V	E	R	Y		'\0'				

word2:

G	O	O	D		'\0'		

strcat(word1,word2);

The execution of the above statement will result in

word1

V	E	R	Y		G	O	O	D	‘\0’

```
   #include<stdio.h>
   #include<string.h>
   void main()
   {
   char str1[31]="Shree",str2[32]="devi";
   printf("\nFirst String is %s\n",str1);

  printf("Second String is%s\n",str2);
   strcat(str1," ")
   strcat(str1,str2);
   printf("Resultant String %s\n",str1);
}
```

Output:

```
First String is Shree
Second String is devi
Resultant String Shree devi
```

strcmp() Function:

This function compares two strings identified by arguments and has value 0 if they are equal. If they are not, it has the numeric difference between the first non-matching characters in the string. It takes the following form

strcmp(string1,string2);

where string1 and string2 may be string variables or string constants. E.g.: **strcmp(“their”, “there”);**

The above statement will return a value -9 which is a numeric difference between ASCII “i” and ASCII “r”. If the value is negative, it means that string1 is alphabetically above string2.

Program:

```
/*Use of Strcmp Function*/
#include<stdio.h>
#include<string.h>
void main(void)
{
char
str1[31],str2[31];
int value;
printf("\n Enter string 1"); gets(str1);
printf("Enter Second String"); gets(str2);
value=strcmp(str1,str2);
if(value>0)
printf("string %s comes after %s in dictionary order\n"str1,str1);
else if(value<0)
printf("string %s comes before %s in dictionary order\n"str1,str2);
else
printf("Both Strings are same\n");
}
```

Output:

```
Enter string 1 Mangalore
Enter Second String Bangalore
Mangalore comes after Bangalore in dictionary order
```

strcpy() Function:

This function works almost like a string-assignment operator. The general format of this function is

strcpy(string1,string2);

The above statement assigns the content of string2 to string1. String2 may be character array or a string constant.

E.g.:

strcpy(city, "DELHI");

This statement will assign the string "DELHI" to string variable city.

If city1= "DELHI" and city2= "CHENNAI". The statement strcpy(city1,city2); will assign the contents of string variable city2 to city1. The size of array city1 should be large enough to contain the contents of city2.

Program:

```
#include<stdio.h>
#include<string.h>
void main()
{
charstr[31]="Shree",str1[32]="shree";
printf("\nFirst String is %s\n",str1);
printf("SecondStringis%s=\n",str2);
strcpy(str1,str2);
printf("Resultant String %s\n",str1);
}
```

Output:

```
First String is Shree
Second String is devi
Resultant Shree devi
```

strlen() Function:

This function counts and returns the number of characters in the specified string. The general format of this function is

n=strlen(string);

where n is an integer variable which receives the value of the length of the string. The argument in this function can be a string constant also. The counting ends when the first null character is en countered.

EXAMPLE :

n=strlen("MANGALORE");

The above statement counts the number of characters in the given string constant and is assigned to n i.e. 9 is assigned to n.

city="DELHI";
strlen(city);

The above statement assigns 5 to the variable m.

Program:

```
#include<stdio.h>
#include<string.h>
void main()
{
charstr[31];
intlen;
printf("\nEnter any String");
gets(str);
len=strlen(str);
printf("\nNumber of Character in%s=%d\n",str,len);
}
```

Output:

```
Enter any String
Jahnavi
Number of Character in Jahnavi=7
```

Table of Strings:

A list of names in a class can be treated as a table of strings and a two dimensional character array can be used to store the entire list. For example, a character array student[30][15] can be used to store a list of 30 names, each of length not more than 15 characters.

Consider the following character array declaration: static char city[][]

```
{
  "Chennai",
  "Madgaon",
  "Mysore",
  "Bombay",
  "Kolkatta"
};
```

The above declaration is stored as follows:

C	H	e	n	n	a	I		
M	A	d	g	a	o	N		
M	Y	s	o	r	e			

B	O	m	b	a	y			
K	O	l	k	a	t	T	A	

In order to access the ith city in the list, we write city[i-1]. Hence city[0] denotes “Chennai”, city[1] denotes “Madgaon” and so on. This shows that once an array is declared as a two-dimensional, it can be used like a one dimensional array for further manipulations. The table can be treated as a column of strings

UNIT IV
CHAPTER 4: User-defined functions

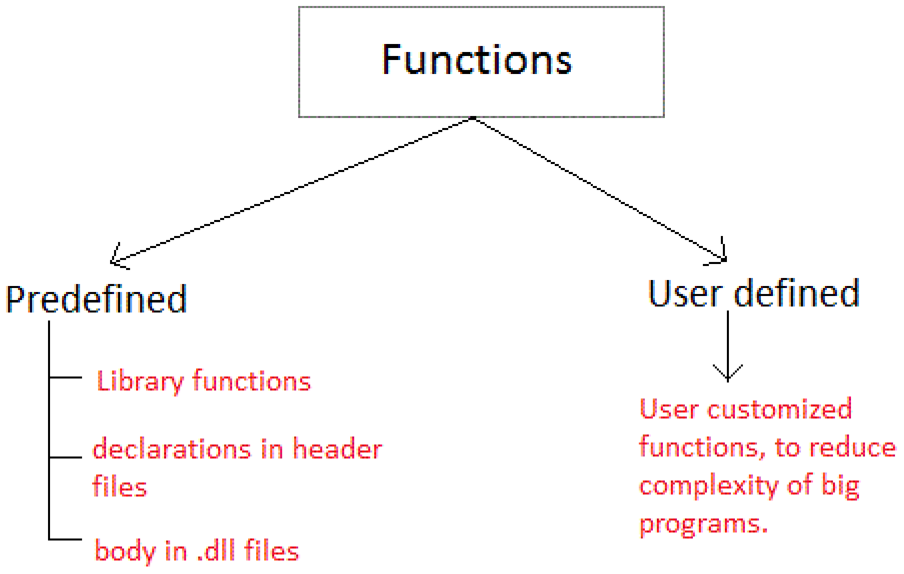
Introduction

There are many situations where we might need to write same line of code for more than once in a program. This may lead to unnecessary repetition of code, bugs and even becomes boring for the programmer. So, C language provides an approach in which you can declare and define a group of statements once in the form of a function and it can be called and used whenever required.

A **function** is a block of code that performs a particular task.These functions defined by the user are also know as **User-defined Functions**

C functions can be classified into two categories,

1. **Library functions**
2. **User-defined functions**


Functions
Predefined
Library functions
declarations in header files
body in .dll files
User defined
User customized functions, to reduce complexity of big programs.

Library functions are those functions which are already defined in C library, eg. strcat(), pow() , strlen() etc.

A **User-defined functions** on the other hand, are those functions which are defined by the user at the time of writing program. These functions are made for code reusability and for saving time and space.

Benefits of Using Functions

1. It provides modularity to your program's structure.
2. It makes your code reusable. You just have to call the function by its name to use it, wherever required.
3. In case of large programs with thousands of code lines, debugging and editing becomes easier if you use functions.
4. It makes the program more readable and easy to understand.

Function prototype

A function prototype is simply the declaration of a function that specifies function's name, parameters and return type. It doesn't contain function body.

A function prototype gives information to the compiler that the function may later be used in the program.

The function prototype is not needed if the user-defined function is defined before the main() function.

Syntax of function prototype

```
returnType functionName(type1 argument1, type2 argument2,...);
```

Function definition

Function definition contains the block of code to perform a specific task.

Syntax of function definition

```
return type functionName(data-type  argument1, data-type  argument2, ...)
{   Local Variable Declarations

    Execution statement(s)

   [return  value/expression]
}
```

Where .

Return data type- type of the return value by the functions.

If nothing is returned to the calling function, then data type is **void.** Function_name - It is the user defined function name.

Argument(s) - The argument list contains valid variable name separated by commas. .

return statement- It is used to return value to the calling function. A function can have multiple return statements . But only one return statement is executed (A function can return only single value). Execution of return statement causes execution to be transferred back to calling function.

When a function is called, the control of the program is transferred to the function definition. And, the compiler starts executing the codes inside the body of a function.

Calling a Function

A function can be accessed or called by specifying its name, followed by a list of arguments enclosed in parentheses and separated by commas. If the function call does not require any arguments, an empty pair of parentheses must follow the function name.

Function parameters

Function parameters are the means of communication between the calling and the called functions. They can be classified into formal parameters and actual parameters. The formal parameters are the parameters given in the function declaration and function definition. The actual parameters(commonly called arguments), are specified in the function call.

```
main ( )

{

   ---------

   ---------

                actual parameters

  function1(a1, a2,  a3...............am);   /* function call*/

  ---------

  ---------

}



function1  (fI,   f2,    f3.............fn)     /* called function*/

{               formal parameters
```

```
  --------
  --------
}
```

Arguments matching between the function call and the called function.

The actual and formal arguments should match in number, type and order. The values of actual arguments are assigned to the formal arguments on a one to one basis starting with the first argument as shown in figure. In case, the actual arguments are more than the formal parameters (m > n), the extra actual arguments are discarded. But if the actual arguments are less than the fomal arguments, the unmatched formal arguments are initialized to some garbage values. Any mismatch in data type may also result in passing of garbage values.

The formal parameters must be valid variable names, but the actual arguments may be variable names, expressions or constants. The variables used in actual arguments must be assigned values before the function call is made.

Example Program 1

```
 # include <stdio.h>
int Add(int x, int y);  /*function prototype or function declaration*/
main()
    {
       int a, b, sum;
       printf("enter two number");
       scanf("%d %d", &a, &b);
       sum=Add(a,b );
       printf("sum of %d and %d is %d", a, b, sum);
     }/*End of the main*/


int Add (int x, int y) /*Function to add two numbers */
{
  return (x + y);
}
```

Type of User-defined Functions in C (Categories of UDF)

There can be 4 different categories of user-defined functions, they are:

1. Function with no arguments and no return value
2. Function with no arguments and a return value
3. Function with arguments and no return value
4. Function with arguments and a return value

Function with no arguments and no return value

Such functions can either be used to display information or they are completely dependent on user inputs.
Below is an example of a function, which takes 2 numbers as input from user, and display which is the greater number.

```
#include<stdio.h>
void greatNum();       // function declaration
int main()
{
   greatNum();       // function call
   return 0;
}

void greatNum()       // function definition
{
   int i, j;
   printf("Enter 2 numbers that you want to compare...");
   scanf("%d%d", &i, &j);
   if(i > j) {
      printf("The greater number is: %d", i);
   }
   else {
      printf("The greater number is: %d", j);
   }
}
```

Function with no arguments and a return value

We have modified the above example to make the function greatNum() return the number which is greater amongst the 2 input numbers.

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

```
int greatNum();       // function declaration

int main()
{
    int result;
    result = greatNum();        // function call
    printf("The greater number is: %d", result);
    return 0;
}

int greatNum()        // function definition
{
    int i, j, greaterNum;
    printf("Enter 2 numbers that you want to compare...");
    scanf("%d%d", &i, &j);
    if(i > j) {
        greaterNum = i;
    }
    else {
        greaterNum = j;
    }
    // returning the result
    return greaterNum;
}
```

Function with arguments and no return value

We are using the same function as example again and again, to demonstrate that to solve a problem there can be many different ways.

This time, we have modified the above example to make the function greatNum() take two integer values as arguments, but it will not be returning anything.

```
#include<stdio.h>
void greatNum(int a, int b);       // function declaration
int main()
{
    int i, j;
    printf("Enter 2 numbers that you want to compare...");
    scanf("%d%d", &i, &j);
    greatNum(i, j);        // function call
    return 0;
}

void greatNum(int x, int y)        // function definition
{
    if(x > y) {
        printf("The greater number is: %d", x);
    }
    else {
        printf("The greater number is: %d", y);
    }
}
```

Function with arguments and a return value

This is the best type, as this makes the function completely independent of inputs and outputs, and only the logic is defined inside the function body.

```
#include<stdio.h>
int greatNum(int a, int b);        // function declaration
int main()
{
    int i, j, result;
    printf("Enter 2 numbers that you want to compare...");
    scanf("%d%d", &i, &j);
    result = greatNum(i, j); // function call
    printf("The greater number is: %d", result);
    return 0;
}

int greatNum(int x, int y)        // function definition
{
    if(x > y) {
        return x;
    }
    else {
        return y;
    }
}
```

Nesting of Functions

C language also allows nesting of functions i.e to use/call one function inside another function's body. We must be careful while using nested functions, because it may lead to infinite nesting.

```
function1()
{
    // function1 body here
        function2();
        // function1 body here
}
```

If function2() also has a call for function1() inside it, then in that case, it will lead to an infinite nesting. They will keep calling each other and the program will never terminate.
Not able to understand? Lets consider that inside the main() function, function1() is called and its execution starts, then inside function1(), we have a call for function2(), so the control of program will go to the function2(). But as function2() also has a call to function1() in its body, it will call function1(), which will again call function2(), and this will go on for infinite times, until you forcefully exit from program execution.

Functions With Arrays

Arrays can be passed as an arguments to functions. To pass an array to a function, the array name must appear by itself, without brackets or subscripts, as an actual argument within the function call. The corresponding formal argument is written in the same manner, though it must be declared as an array within the formal argument declarations. When declaring a one-dimensional array as a formal argument, the array name is written with a pair of empty square brackets. The size of the array is not specified within the formal argument declaration.

For example, the call

Max(A,N); Will pass all the elements contained in the array A of size N. The Max function header can be

```
int    Max(int Array[], int size)
```

or

```
int Max( Array,size)
 int Array[], size;
```

The function Max is declared to take two arguments, the array name and the size of the array to specify the number of elements in the array,The declaration of the formal argument Array is made as follows.

```
        int Array[];
```

The pair of brackets informs the compiler that the argument as an array of numbers.

Example program 5

```
/*Function program to find the maximum number from the array of N given elements*/
#include<stdio.h>
int Max(int a[ ],int n); /* declaration part */
main( )
{
  int num[100],m,i;
  clrscr( );
/* input part */
printf("\nEnter the number of elements :");
scanf("%d" ,&m);
printf("Enter %d elements\n",m);
for(i=0;i<m;i++ )
   scanf("%d",&num[ i ]);
/* processing and output part * /
clrscr( );
printf("Entered elements are...\n");
for(i=0;i<m;i++ )
   printf("%d\n",num[ i ]);
printf("\nThe maximum element is %d",Max(num,m));
getch( );
}
```

```
int Max(int a[ ],int n)
{
int x,i;
 x=a[0];
 for(i=1 ;i<n;i++)
 if (x < a[ i ])
  x=a[ i ];
return x;
}
```

When an array is passed to a function, the value of the array elements are not passed to the function. Rather, the array name is interpreted as the address of the first element. i.e., the address of the memory location containing the first array element. This address is assigned to the corresponding formal argument when the function is called. The formal argument therefore becomes a pointer to the first array element. Arguments passed in this manner are said to be passed by reference. .

When a reference is made to an array element within the function, the value of the element's subscript is added to the value of the pointer to indicate the address of the specified array element. Therefore, any array element can be accessed from within the function. Moreover, if an array element is altered within the function, the alteration will be recognized in the calling portion of the program.

The return statement cannot be used to return an array (return statement can return only a single-valued expression to the calling portion of the program). Therefore, if the elements of an array are to be passed back to the calling portion of the program, the array must either be defined as an external array whose scope includes both the function and the calling portion of the program, or it must be passed to function as a formal argument.

Example program 6

```
/*Function program to sort N given elements using bubble sort*/
#include<stdio.h>
void sort(int a[ ],int n);
main( )
{
/* declaration part */
int b[10],m,i;
clrscr( );
/* input part */
printf("\nEnter the number of terms :");
scanf("%d" ,&m);
printf("Enter %d terms\n",x);
for(i=0;i<m;i++ )
scanf("%d",&b[i]);
  /* processing and output part */
  clrscr( );
 printf("Entered numbers are \n");
for(i=0;i<n ;i++)
printf("%d\n" ,b[i]);
sort(b,x);
printf("After sorting the numbers are\n");
for(i=0;i<m;i++)
printf("%d\n" ,a[ i ]);
getch( );
 }
void sort(int a[ ],int n)
 {
 int i,j,temp;
  for(i=0;i<n;i++ )
   for(j=0;j<n-i-l ;j++)
    {
      if(a[j]>a[j+ 1])
    temp=a[j] ;
    a[j]=a[j+l];
    a[j+1 ]=temp;
  }
}
```

Recursion

Recursive functions are functions that call themselves, so a **recursion** is process of calling function by itself.

Example program 7

```
/*Recursive function program to find the factorial of a number*/

#include<stdio.h>

int fact(int n);

main( )

(

 int num;

 clrscr( );

 printf("\nEnter a number :");

   scanf("%d" ,&num);

/* processing and output part */

   clrscr( );

   printf("Entered number: %d",num);

   printf("\nFactorial of the entered number: %d", fact(num));

   getch( );

}

int fact(int num)

{

if(num==0)

      return (1);

else

     return (num * fact(num-1);
```

}

Suppose the definition fact is called with the value 2, then the value of num in the function fact is 2, and the condition in the if statement is false, and the function gets called again with the value num - 1, which is 1. In the second invocation of fact, the value of num is 1. The condition in the if statement is again false, and a third invocation of fact results, passing 0 to it. In the third invocation, the value of num 0, and the condition of fact returns 1 to the second invocation.

Hence, the statement

```
return (num * fact (num-1);
```

Multiplies 1 by 1 and causes the number 1 to be returned to the first invocation; where the value of num was 2. This brings back to the statement

```
return (num * fact(num-l));
```

Which is now executing in the first invocation. The return value of the second invocation, 1 is multiplied by 2 and the value 2 is returned. Two important conditions must be satisfied by any recursive function:

i) Each time when a function calls itself, it must be closer, in some sense to a solution.

ii) There must be a decision criterion for stopping the process or computation.

For the function factorial, each time when the function calls itself, its argument is decremented by one. The stopping criterion is the if statement that checks for the zero argument.

Example Program 8

/* Function program to calculate X^n using recursive function*/

```
#include<stdio.h>

float power(float x,int n);

 main( )

{

float a;

int b;

clrscr( );

/* input part */

printf("Enter the values of X and n :");
scanf("%f, %d",&a,&b);
 /* processing and output part */
clrscr( );
 printf("Base = %.Of',a);
 printf("\nExponent = %d",b);
 printf("\n%f  to the power   %d is = %f",a,b,power(a,b));
   getch( );
}
float power(float x,int n)
{
if (x==0)
     return 0.0;
else


if (n==0)
   return 1.0;
else
  if (n>O)
         retum(power(x,n- 1 )*x);
  else
         retum(power(x,n+ 1 )/x);
}
```

Example Program 9

```
/*Recursive function program to generate first N terms of fibonacci series */
#include<stdio.h>
long fibo(int j);
main( )
{
 int n,i;
 clrscr( );
/* input part */
printf("\nEnter the number of terms :");
scanf("%d" ,&n); /* processing and output part */
  clrscr( );
  printf("The first %d fibonacci numbers are...\n",n);
 fore i=0;i<n;i++)
     printf("%ld ",fibo(i));
   getch( );
}
long fibo(int j )
{
int f;
if (j= =0)
       return 0;
else if (j= =1)
       return 1;
else
      return (fibo(j-l) + fibo(j-2));
}
```

UNIT 4
CHAPTER 5: Storage classes

The variables in 'C' can have anyone of the following **four** storage classes:

i) **Automatic variables**
ii) **Register variables**
iii) **Static variables**
iv) **External variables**

The scope of the variable determines over what part of the program a variable is actually available for use. The variables are also classified depending on the place of their declaration, as **internal** (local) or **external** (global). Local or internal variables are those which are declared within a particular function, whereas external variables are declared outside of any function.

Automatic Variables

The variables in main() are private or local to main(). Because they are declared within main, no other function can have direct access to them. The same is true of the variables in other functions. Each local variable in a function comes into existence only when the function is called and disappeared when the function is exited. Such variables are usually known as **automatic** variables.We can make use of the optional keyword in the declaration, auto for **Example**

Auto int x;

auto has been the default class for all the variables we have used so for. The value of the automatic variables cannot be changed in some other function in the program. So, we can declare and use the same variable name in different functions in the same program.

Example Program 10

```
#include<stdio.h>
int Add (int a,int b);
 main( )
{
auto int a=10,b=15,c;
clrscr( );
c= a+b;
printf("c= %d", c);
c= Add(a,b);
printf("\nc= %d", c);
  }
int Add( int a, int b)
   {
     int c;
    c= a+b;
    return c;
  }
```

Register Variables

Register variables are local variables (similar to automatic variables) except that they are placed in machine registers. A register declaration advises the compiler that the variable in question will be heavily used. The idea is that register variables are to be placed in machine registers, which may result in smaller and faster programs. But compilers are free to ignore the advice.

The register declaration looks like

```
        register int i;
```

Then the compiler may try to place the variable 'i' in a machine register if available.

The register declaration can only be applied to automatic variables and the parameters of a function. Since only a few variables can be placed in the register, 'C' will automatically convert register variables into auto variables once the limit is reached. Loop indices, accessed more frequently, can be declared as register variables. For example 'index' is a register variable in the program given below.

Example Program 11

```
# inc1ude<stdio.h>
main( )
 {
   register index, Sum = 0;
for (index== 1; index<==100; index++)
Sum == Sum + index;
 printf(" Sum == %d", Sum);
}
```

Finally one cannot assign large or aggregate data types, such as doubles or arrays to registers and the & operator cannot be applied to register variables.

External Variables

. External variables are defined outside of any function, and are thus available to many functions. Hence, they are also known as global variables. By default, external variables and functions have the property that all references to them by the same name.

Because external variables are globally accessible, they provide an attention to function arguments and return values for communicating data between functions. Any function may access an external variable by referring to it by name. If a number of variables must be shared among functions, external variables are more convenient and efficient than long argument lists. External variables are also useful because of their greater scope and lifetime. Automatic(local) variables are internal to a function; they come into existence when the function is entered, and disappeared (destroyed) when it is left. External variable on the other hand, are permanent, so they retain values from one function invocation to the next.

Thus if functions must share some data, yet neither calls the other, it is often most convenient if the shared data is kept in external variables rather than passed in and out via arguments.

One important feature of global variable is that if a function has a local variable, of the same name as a global variable, the local variable has precedence over the global variable. Analogously, a variable declared within a block has precedence over an external variable of the same name.

In order to be able to use an externally declared variable inside' a function, such a variable may explicitly redeclared inside the function with the **extern** qualifier, for example:

extern int X;

The external keyword tells the 'C' compiler and linker that the variable thus qualified' is actually the one of the same name defined externally; this is the one that will be used in the function. New storage is not created for it, because it refers to a variable that already exists. Thus, the important distinction between the terms definition and declaration is that, a definition creates a variable and allocates storage. A declaration describes a type but does not reserve space. Precisely for this reason an initial value cannot be specified with an extern declaration: initialization means that a value is assigned at the time that storage is allocated; because storage is not allocated by the extern declaration, it .cannot include an initialization. Therefore, inside a function the following statement is unacceptable.

extern int x = 5 ; /* WRONG */

Example Program 12

```
#include <stdio.h>

int Sum;

void Inc( );

main( )

{

    int a,b;

    clrscr( );

    priiltf("Enter two numbers");

    scanf("%d %d", &a,&b);

    Sum = a+b;

printf("Sum of %d and %d is %d", a,b,Sum);

Inc( );

printf("\nAfter incrementing the value of Sum is %d",Sum);

getch( );

}
```

```
 void Inc( )
{
Sum ++;
}
```

In the above program Sum is used in the function Inc( ). Since Sum has been declared above all the functions, it is available to each function without having to pass Sum as a function argument. Once a variable has been declared as global, any function can use it and change its value.

Static Variables

Another C storage declaration is static. The keyword static is used to declare variables which must not lose their storage locations or their values when control leaves the functions or blocks where in they are defined. In C, static variables retain their values between function calls. The initial value assigned to a static variable must be a constant, or an expression involving constants. But static variables are by default initialized to 0, in contrast to autos.

For example

```
        Static int i;
```

Initalises variable i to 0

Example Program 13

```
 #include<stdio.h>
void Display(void);
main( )
{
int i;
clrscr( );
for ( i=1; i<=4; i++)
Display( );
}
void Display(void)
   {
  static int x= 0;
```

```
   x++;
  printf("\nx=%d" ,x);
   }
```

output:

```
x=1
x=2
x=3
x=4
```

A static variable is initialized only once, when the program is compiled. It is never initialized again. During the first call to Display( ), x is incremented to I. Because x is static, this value retains and therefore, the next call adds another I to x giving its value equal to 2. The value of x becomes 3 when third call is made and 4 when fourth call is made.

In case, if the variable x is of type auto then the value of x would be x=1 x=1 x=l x=1. This is because each time Display is called, the auto variable x is initialized to zero. When the function terminates, its value i.e. 1 is lost.

UNIT 5

CHAPTER 6: Structures and unions:

INTRODUCTION

Arrays provide a means to homogenous data items into a single named unit. But most of the applications require the grouping together of heterogeneous data items. Such an entity is called **structure** and the related data items use in it are referred to as members. Thus a single structure might contain integer elements, floating point elements and character elements. Pointers, arrays and other structures can also be included as elements with in a structure.

DEFINING A STRUCTURE

Every structure must be defined and declared before it appears in a C program .The general form of structure definition and declaration is as follows,

```
struct tagname
{
  data-type member_1 ;
```

```
            data-type member_2;
            -----------
            -----------
        data-type member_n;
            } ;
```

In this declaration struct is a required keyword ,tagname is a name of the structure and member_1 ,member_2, ...member_n are individual member declarations. The individual members can be ordinary variables, pointers, arrays or other structures. The member names with in a particular structure must be distinct from one another though a member name can be same as the name of a variable defined outside of the structure.

Example

```
struct Student
          {
               int Regno;
               char Name[20);
               int m1,m2,m3,Total_marks;
                float Avg;
          };
```

This structure named Student (i.e, the tag is student) contains seven members, an integer variable Regno ,a 20-element character array (Name [20]). integer variables m1,m2,m3,Total_marks and a floating type variable Avg.

The general form of declaring structure variable is

struct structure-type variable- list;

The following declaration declares student to be of type Std.

```
struct Student Std;
```

THE DOT OPERATOR

In order to gain access to a member of a structure, the dot operator "." is used. That is a structure member can be accessed by writing.

structure_name. member_name

Where structure_name refers to the name of a structure *type* variable and member _name refers to the name of member with in the structure. The period (.) is an operator and it is a member of the highest precedence group and it groups from left to right.

The members of a structure can be initialized to constant values by enclosing the values to be assigned in braces after the structures definitions.

Example

```
 struct student std = {1001,"Kiran",90,90,90,270,90};
```

The individual values are separated by commas and enclosed between braces. The closing brace is followed by semicolon.

1l1is structure named Student (i.e, the tag is student) contains four members, an integer variable Regno ,a 20-element character an'ay (Name[20]). an integer variable Total_marks and a floating type variable Avg.

Example Program 1

```
 #include<stdio.h>
#include<conio.h>
void main( )
{
struct employee {
                    long int code;
                    char name[20];
                     float salary;


                  } ;
struct employee emp;
clrscr( );
printf(" Enter employee code:");
scanf("%d" ,&emp.code);
fflush(stdin);
printf("Enter employee name:");
gets(emp.name);
fflush(stdin);
printf("Enter salary:");
scanf("%f' ,&emp.salary);
clrscr( );
printf("EMPLOYEE INFORMATION SYSTEM\n\n\n");
printf("CODE          NAME          SALARY\n");
printf("%d\t %s\t %f\n",emp.code,emp.name, emp.salary);
 getch( );
 }
```

A structure may be defined as a member of another structure. In such cases ,the declaration of the embedded structure must appear before the declaration of the outer structure.

<u>Example</u>

```
        struct date
                {int day;
                 int month;
                 int year;
               } ;
       struct student
       {
        int Regno;
        char Name[20];
        struct date DOB
      } std;
```

or

it can also be written as

```
           struct Student
        {
                int Regno;
                 char Name[20];
                struct date
          struct date
               {int day;
                int month;
                int year;
              }DOB ;
        }  ;
```

If a structure member is itself a structure then a member of the embedded structure can be accessed by writing

```
variable. member.submember
```

Where member refers to the name of the member within the outer structure, and submember refers to the name of the member within the embedded structure.

In the above example the last member of std is std.DOB which is itself a structure of type date. To access the year of date of birth, we have to write as

```
std.DOB.year;
```

ARRAY OF STRUCTURES

Whenever some structure that is to be applied to a group of people or items, array of structures can be used.

Example

```
struct Stud
{
int Regno;
char Name[20];
int mark 1, mark2, mark3, Total;
float Avg;
};
struct Stud Student[10];
```

Here Student is an array of 10 elements of type Stud. Thus Student [0] will contain the first set of values, Student[1] will contain the second set of values and so on.

TYPE DEFINITION: typedef STATEMENT

The type definition statement is often used for defining new data type involving structures. Since it eliminates the need to repeatedly write struct tag whenever a structure is referenced. A new data type representing the structure is declared using the typedef keyword. As a result the structure can be referenced using the type name alone.

In general terms a user defined structure can be written as

```
typedef struct

{

 data-type member_1 ;

data-type member_2;

-----------

-----------

data-type member_n;

} new_type;
```

where new_type is the user defined structure type. Structure variables can then be defined in terms of the new data type.

The usage of the typedef statement is illustrated below:

```
typedef struct Stud

{

 int Regno;

char Name[20];

int Total;

float Avg;

} Student;
```

Here Stud is the tag while Student is the name of the new type. Instead of declaring the variable using the tag, for example

```
struct Stud Std1;
```

we can declare it as

```
                              student Std1 ;
```

That is the type Student does not require the struct keyword. Since variables of the new structure can be created using the type name Student , the tag Stud can be eliminated from the structure declaration.

UNION

Union like structures, contains members whose individual data type may differ from one another. The difference between union and structure the member that compose a union all share the same storage area within the computer's memory. Where as in structures, each member has its own storage location. This implies that, although a union may contain many members of different types, it can handle only one member at a time. They are useful for applications involving multiple members where values need not be assigned to all of the members at anyone time. Thus unions are used to conserve memory.

The general form of union is

```
union tagnmae
 {
Data-type member__1;
Data-type member_2;
-----------
Data- type member n;
};
```

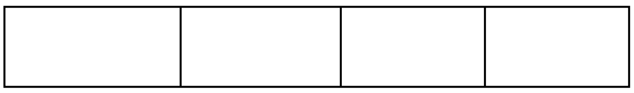
Where union is a required keyword and the tag is optional.

A union may be initialized by value of the type of its first member. If a union has been assigned a value of certain type then it is not correct to attempt to retrieve from it a value of a different type.

Example:

```
union Item
{ int x;
  float y;
  char z;
} Alpha;
```

This declares a variable Alpha of type union Item. The union contains three members each with a different data type. But it is possible to use only one of them at a time. This is due to the fact that only one location is allocated for a union variable irrespective of size.



```
----z--------

--------------x-----------

 ---------------------------y---------------------
```

Fig. Sharing of storage locations by Union members

The compiler allocates a piece of storage that is large enough to hold the largest variable type in the union. In the declaration above the member (float type) requires 4 bytes which is the largest among the members. Figures shows how all the three variables share the same address.

To access union members we have to use dot operator .

Example :-

Alpha.x

Alpha.y

Alpha.z

BIT FIELDS

So far, we have been using integer fields of size 16.bits to store data. There are occasions where data items require much less than 16 bits space. In such cases, we waste memory space. Fortunately, C permits us to use small bit fields to hold data items and thereby to pack several data items in a word of memory. Bit fields allow direct manipulation of string of a string of preselected bits as if it represented an integral quantity.

A bit field is a set of adjacent bits whose size can be from 1 to 16 bits in length. A word can therefore be divided into a number of bit fields. The name and size of bit fields are defined using a structure. The general form of bit field definition is:

struct *tag-name* {

data-type name1: *bit-length;*

data-type name2: *bit-length;*

……………..

data-type nameN: *bit-length;*

}

The *data-type* is either int or unsigned int or signed int and the *bit-length* is the number of bits used for the specified name. Remember that a signed bit field should have at least 2 bits (one bit for sign). Note that the field name is followed by a colon. The *bit-length* is decided by the range of value to be stored. The largest value that can be stored is 2^n-1, where n is bit-length.

The internal representation of bit fields is machine dependent. That is, it depends on the size of int and the ordering of bits. Some machines store bits from left to right and others from right to left. The sketch below illustrates the layout of bit fields, assuming a 16-bit word that is ordered from right to left.

There are several specific points to observe:

1. The first field always starts with the first bit of the word.

2. A bit field cannot overlap integer boundaries. That ,is, the sum of lengths of all the field! in a structure should not be more than the size of a word. In case, it is more, the overlapping field is automatically forced to the beginning of the next word.

3. There can be unnamed fields declared with size.

Example:

Unsigned : *bit-length;*

Such fields provide padding within the word.

4. There can be unused bits in a word.

5. We cannot take the address of a bit field variable. This means we cannot use
scanf to read values into bit fields. We can neither use pointer to access the bit
fields.

6. Bit fields cannot be arrayed.

7. Bit fields should be assigned values that are within the range of their size. If we
try to assign larger values, behaviour would be unpredicted. .

Suppose, we want to store and use personal information of employees in compressed form This can be done as follows:

```
struct personal

{

unsigned sex:1

unsigned age: 7

unsigned m_status:1

 unsigned children:3
```

unsigned:4

} emp;

This defines a variable name emp with four bit fields. The range of values each field could have is as follows:

Bit field	*Bit length*	*Range of values*
sex	1	0 or 1
age	7	0 to 127 (27_1)
m_status	r	0 or 1 .
children	3	0 to 7 (23_1)

Once bit fields are defined, they can be referenced just as any other structure-type data item would be referenced. The following assignment statements are valid.

emp.sex = 1;

emp.age = 50;

Remember, we cannot use scanf to read values into a bit field. We may have to read into a temporary variable and then assign its value to the bit field. For example:

scanf("%d %d", &AGE,&CHILDREN);

emp.age = AGE;

emp.children = CHILDREN;

One restriction in accessing bit fields is that a pointer cannot be used. However, they can be used in normal expressions 1ike any other variable. For example,

sum = sum + emp.age;

if(emp.m_status)

printf("%d\n", emp.age);

are valid statements.

UNIT 5

CHAPTER 7: Pointers

INTRODUCTION

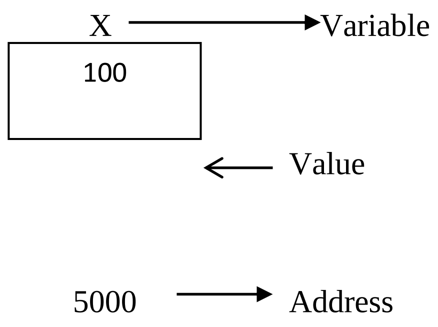
As you know, computers use their memory for storing the instructions of a program, as well a the values of the variables that are associated with it. The computer's memory is a sequential collection of 'storage cells'. Each cell, commonly known as a *byte,* has number called *address* associated with it. Typically, the addresses are numbered consecutively starting from zero. The last address depends on the memory size. A computer system having 64K memory will have its last address as 65,535.

Whenever we declare a variable, the system allocates, somewhere in the memory, appropriate location to hold the value of the variable. Since, every byte has a unique address number, this location will have its own address number. Consider the following statement:

int x = 100; I

This statement instructs the system to find a location for the integer variable x and store the value 100 in that location. Let us assume that the system has chosen the address location 5000 for quantity. We may represent this as shown in

Fig.



During execution of the program, the system always associates the name **x** with the address 5000. We may have access to the value 100 by using either the name **x** or the address 5000. Since memory addresses are simply numbers, they can be assigned to some variables which can be stored in memory, like any other variable. Such variables that hold memory addresses are called *pointers.* **A pointer is, therefore, nothing but a variable that contains an address which is a location of another variable of same type.**

Remember, since a pointer is a variable, its value is also stored in the memory in another location. Suppose, we assign the address of **x** to a variable p. The link between the variables p and **x** can be visualized as shown in Fig below The address of p is say 5050.

Variable	Value	Address
x		5000

$\boxed{100}$

$\boxed{5000}$

Since the value of the variable p is the address of the variable **x,** we may access the value of **x** by using the value of p and therefore, we say that the variable p 'points' *to* the variable **x.** Thus, p gets the name 'pointer'.

Accessing the Address Of A Variable

The actual location of a variable in the memory is system dependent and therefore, the address of a variable is not known to us immediately. The address can be obtained with the help of the operator & available in C. . The operator & immediately preceding a variable returns the address of the variable associated with it. For example, the statement

```
p = &x;
```

will assign the address 5000 (the location of x) to the variable p. The & operator

be remembered as **'address of'**

The & operator can be used only with a simple variable or an array element.

Following are illegal.

```
X=&100;
Y=&(x+y);
```

If x is an array, then expressions such as

```
&x[0] and &x[i+3]
```

are valid and represent the addresses of 0th and (i+3)th elements of x.

Example

Write a program to print the address of a variable along with its value.

```
 main( )
{


char a; int x;
float p, q;
 a = 'A';
 x = 125;
 P = 10.25, q = 18.76;
printf("%c ,is stored at addr %u.\ n", a, &a);
printf("%d 'is stored at addr%u. \ n", x, &x);
printf("%f is stored at addr %u. \n", p, &p);
printf("%f is stored at addr %u.\n", q, &q);
}
```

DECLARING AND INITIALIZING POINTERS

In C, every variable must be declared for its type. Since pointer variables contain addresses that belong to a separate data type, they must be declared as pointers before we use them. The declaration of a pointer variable takes the following form:

***data type* *pt_name;**

This tells the compiler three things about the variable pt_name.

1. The asteris (*) tells that the variable pt_name is a pointer variable.
2. pt_name needs a memory location.
3. pt_name points to a variable of type *data type.*

For example,

```
int *p;
```

declares the variable p as a pointer variable that points to an integer data type. Remember that the type int refers to the data type of the variable being pointed to by p and not the type of the value of the pointer. Similarly, the statement

```
float *temp;
```

declares temp as a pointer to a floating point variable. Once a pointer variable has been declared, it can be made to point to a variable using assignment statement such as

```
p = &x;
```

which causes p to point to x. That is, p now contains the address of x. This known as pointer *initialization.* Before a pointer is initialized, it should not be used.

We must ensure that the pointer variables always point to the corresponding type of data For example,

```
float a, b;
int     x,*p;
p       =&a;
a=10;
```

will result in erroneous output because we are trying to assign the address of a float variable to integer pointer. When we declare a pointer to be of int type, the system assumes that an address that the pointer will hold will point to an integer variable. Since the compiler will not detect such errors, care should be taken to avoid wrong pointer assignments. And also assigning an absolute address to a pointer variable is prohibited. The following is wrong.

```
int  *ptr;
ptr = 5368;
```

A pointer variable can be initialized in its declaration itself. For example, .

```
int x, *p = &x;
```

is perfectly valid. It declares x as an integer variable and p as a pointer variable and the initializes p to the address of x. Note carefully that this is an initialization of p, not *p. And also remember that the target variable x is declared first. The statement .

```
int *p = &x, x;
```

is not valid.

ACCESSING A VARIABLE THROUGH ITS POINTER

Once a pointer has been assigned the address of a variable, the question remains as to how to access the value of the variable using the pointer. This is done by using another unary operator * (asterisk), usually known as the *indirection* operator. Consider the following statements:

```
int x,*p, n;
x = 200;
```

p = &x;

n = *p;

The first line declares x and n as integer variables and p as a pointer variable pointing to an integer. The second line assigns the value 200 to quantity and the third line assigns the address of x to the pointer variable p. The fourth line contains the indirection operator *. When the operator * is placed before a pointer variable in an expression (on the right-hand side of the equal sign), the pointer returns the value of the variable of which the pointer value is the address. In this case, *p returns the value of the variable x, because p is the address of x.. The * can be remembered as 'value at address'. Thus the value of n would be 200.

POINTER EXPRESSIONS

Like other variables, pointer variables can be used in expressions. For example, if pl and p2 are properly declared and initialized pointers, then the following statements are valid.

y = *p1 * *p2;

sum = sum +- *p1;

z = 5* - *p2 / *p1;

*p2 = *p2 + 10;

Note that there is a blank space between / and * in the item3 above. The following is wrong.

z = 5* - *p2 /*p1;

The symbol /* is considered as the beginning of a comment and therefore the statement fails.

C allows us to add integers to or subtract integers from pointers, as well as to subtract one pointer from another. pl + 4, p2 - 2 and p1 - p2 are all allowed. If pl and p2 are both pointers to the same array, then p2 – p1 gives the number of elements between pl and p2. We may also use short-hand operators with the pointers.

p1++; --p2; sum += *p2; are all valid.

POINTER INCREMENTS AND SCALE FACTOR

We have seen that the pointers can be incremented like

p1 = p2 + 2; **p1 = p1** + 1; and so on. Remember, however. an expression like p1++;

will cause the pointer **p1** to point to the next value of its type. For example, if **p1** is an integer pointer with an initial value, say 2800, then after the operation **p1 = p1 + 1,** the value of **p1** will be 2802, and not 2801. That is, when we increment a pointer, its value is increased by the length of the data type that it points to. This length is called the ***scale factor***. *T*he lengths of various data types are as follows:

characters 1 byte integers 2 byte floats 4 bytes

long integers 4 bytes doubles 8 bytes

The number of bytes used to store various data types depends on the system and can be found by making use of the sizeof operator. For example, if x is a variable, then sizeof(x) returns the number of bytes needed for the varible.

POINTERS AND ARRAYS

When, an array is declared, the compiler allocates a base address and sufficient amount of storage to contain all the elements of the array in contiguous memory locations. The base address is the location of the first element (index 0) of the array. The compiler also defines the array name as a constant pointer to the first element. Suppose we declare an array x as follows:

static int x[5] = {1, 2, 3, 4, 5};

Suppose the base address of x is 1000 and assuming that each integer requires two bytes, the five elements will be stored as follows:

X[0] x[1] x[2] x[3] x[4]

1	2	3	4	5

Address-> 1000 1002 1004 1006 1000

The name x is defined as a constant pointer pointing to the first element, x[0] and therefore the value of x is 1000, the location where x[O] is stored. That is,

x = &x[O] = 1000

If we declare p as an integer pointer, then we can make the pointer p to point to the array x by the following assignment:

p = x;

This is equivalent to p = **&x[0];**

Now, we can access every value of x using p++ to move from one element to another.

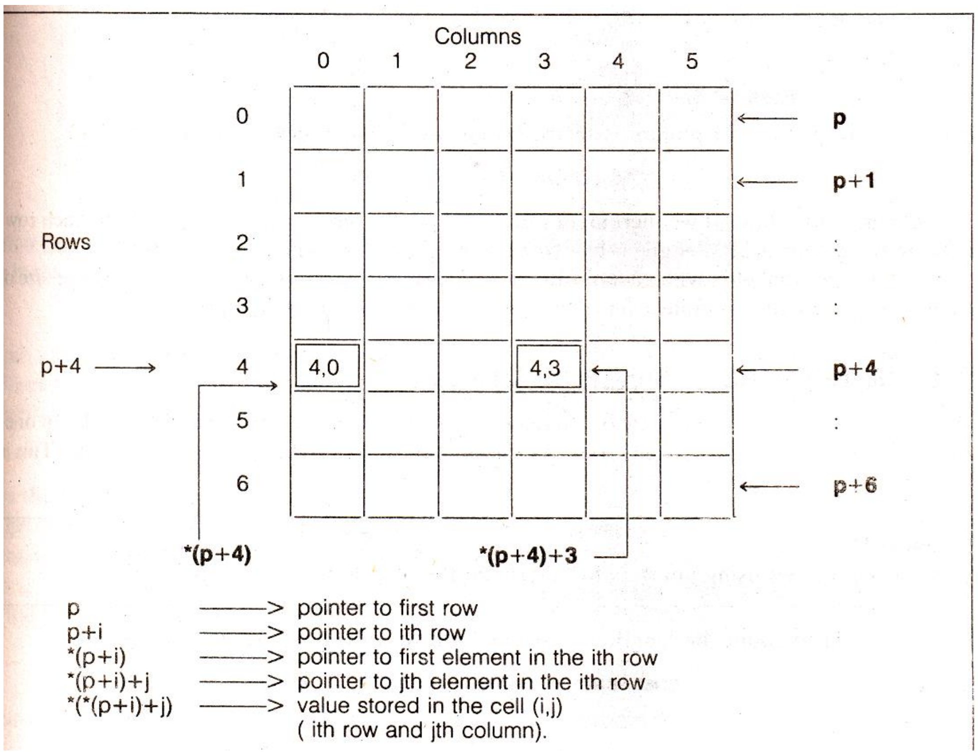
Pointers can be used to manipulate two-dimensional arrays as well. We know that in a one. dimensional array x, the expression

*(x+i) or *(p+i)

represents the element x[i]). Similarly, an element in a two-dimensional array can be represented by the pointer expression as follows:

((a+i)+j) or *(*(p+i)+j)

Figure below illustrates how this expression represents the element a[i)[j]. The base address of the array a is &a[0][0] and starting at this address, the compiler


Columns
0
1
2
3
4
5
Rows
0
1
2
3
4
5
6
p
p+1
:
:
p+4
:
p+6
p+4
4,0
4,3
*(p+4)
*(p+4)+3
p ——> pointer to first row
p+i ——> pointer to ith row
*(p+i) ——> pointer to first element in the ith row
*(p+i)+j ——> pointer to jth element in the ith row
((p+i)+j) ——> value stored in the cell (i,j) (ith row and jth column).

POINTERS AND CHARACTER STRINGS

We know that a string is an array of characters. terminated with a null character. Like in one dimensional arrays. we can use a pointer to access the individual characters in a string. This is illustrated by the example 5.

Example

```
main ()
  {
     char s[10] = "computer";
     char *p = s;      /* pointer p is pointing to string s.   */
     printf("%s",p);    /* the string computer is printed if we print p. */
  }
```

In the following example, we have shown the use of pointers to copy the content of a string into another.

```
#include<stdio.h>
void main ()
{
   char *p = "hello C";
   printf("String p: %s\n",p);
   char *q;
   printf("copying the content of p into q...\n");
   q = p;
   printf("String q: %s\n",q);
}
```

Output

```
String p: hello C
copying the content of p into q...
String q: hello  C
```

Once a string is defined, it cannot be reassigned to another set of characters. However, using pointers, we can assign the set of characters to the string. Consider the following example.

```
#include<stdio.h>
void main ()
{
        char *p = "hello C";

        printf("Before assigning: %s\n",p);

        p = "hello";

        printf("After assigning: %s\n",p);

    }
```

Pointers as Function Arguments

The **call by value** method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the actual arguments.

By default, C programming uses *call by value* to pass arguments. In general, it means the code within a function cannot alter the arguments used to call the function. Consider the function **swap()**definition as follows.

```
/* function definition to swap the values */
void swap(int x, int y) {

   int temp;

   temp = x; /* save the value of x */
   x = y;    /* put y into x */
   y = temp; /* put temp into y */


}
```

Now, let us call the function **swap()** by passing actual values as in the following example −

```
#include <stdio.h>

/* function declaration */
void swap(int x, int y);

  main () {

   /* local variable definition */
   int a = 100;
   int b = 200;

   printf("Before swap, value of a : %d\n", a );
   printf("Before swap, value of b : %d\n", b );

   /* calling a function to swap the values */
   swap(a, b);

   printf("After swap, value of a : %d\n", a );
   printf("After swap, value of b : %d\n", b );



}
```

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :100
After swap, value of b :200
```

It shows that there are no changes in the values, though they had been changed inside the function.

In **call by reference** method address of an argument is passed to the formal parameter. So formal parameters should be pointers.

as in the following function **swap()**, which exchanges the values of the two integer variables using pointers

```
/* function definition to swap the values */
void swap(int *x, int *y) {

   int temp;
   temp = *x;    /* save the value at address x */
   *x = *y;      /* put y into x */
   *y = temp;    /* put temp into y */

   return;
}
```

Let us now call the function **swap()** by passing values by reference as in the following example −

```
#include <stdio.h>

/* function declaration */
void swap(int *x, int *y);

  main ()
{
   int a = 100;
   int b = 200;

   printf("Before swap, value of a : %d\n", a );
   printf("Before swap, value of b : %d\n", b );

   /* calling a function to swap the values.
      * &a indicates pointer to a ie. address of variable a and
   * &b indicates pointer to b ie. address of variable b.
 */
 swap(&a, &b);

 printf("After swap, value of a : %d\n", a );
 printf("After swap, value of b : %d\n", b );

}
```

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :200
After swap, value of b :100
```

It shows that the change has reflected in the function as well, unlike call by value where the changes do not reflect in the called function.

UNIT 5
CHAPTER 8: File Management

Introduction

The input-output functions such as scanf and printf are used to read and write datafrom the terminal (keyboard and screen). This works fine as long as the data is small. However, many real-life problems involve large volumes of data and in such situations, the console oriented I/O operations pose two major problems.

1. It becomes cumbersome and time consuming to handle large volumes of data

   through terminals.

2. The entire data is lost when either the program is terminated or the computer is

   turned off.

It is therefore necessary to have a more flexible approach where data can be stored on the disks and read whenever necessary, without destroying the data. This method employs the concept of *files* to store data. A file is a place on the disk where a group of related data is stored. C supports a number of functions that have the ability to perform basic file operations, which include:

- ✓ naming a file,
- ✓ opening a file,
- ✓ reading data from a file,
- ✓ writing data to a file, and
- ✓ closing a file.

Defining File Pointer And Opening A File

If we want to store data in a file in the secondary memory, we must specify certain things about the file, to the operating system. They include:

1. Filename.
2. Data structure.
3. Purpose.

Filename is a string of characters that make up a valid filename for the operating system. It may contain two parts, a primary name and an optional period with the extension.

Data structure of a file is defined as FILE in the library of standard I/O function definitions. Therefore, all files should be declared as type FILE before they are used. FILE is a defined data type.

When we open a file, we must specify what we want to do with the file. For example, we may write data to the file or read the already existing data.

Following is the general format for declaring and opening a file:

```
FILE *fp;

fp =  fopen("filename", "mode");
```

The first statement declares the variable fp as a "pointer to the data type FILE". As stated earlier, FILE is a structure that is defined in the I/O library. The second statement opens the file named *filename* and assigns an identifier to the FILE type pointer fp. This pointer which contains all the information about the file is subsequently used as a communication link between the system and the program.

The second statement also specifies the purpose of opening this file. The *mode* does this job.*Mode* can be one of the following:

r	open the file for reading only.
w	open the file for writing only.
a	open the file for appending (or adding) data to it.

Note that both the filename and mode are specified as strings. They should be enclosed in double quotation marks. .

When trying to open a file, one of the following things may happen:

1. When the mode is 'writing', a file with the specified name is created if the file does not exist. The contents are deleted, if the file already exists.

2. When the purpose is 'appending', the file is opened with the current contents safe. A file with the specified name is created if the file does not exist.

3. If the purpose is 'reading', and if it exists, then the file is opened with the current' contents safe; otherwise an error occurs.

Consider the following statements:

FILE *p1, *p2;

p1 = fopen("data", "r");

p2 = fopen("results", "w");

The file data is opened for reading and results is opened for writing. In case, the results file already exists, its contents are deleted and the file is opened as a new file. If data file does not exist, an error will occur.

We can also specify **mode** in following form

r+ The existing file is opened to the beginning for both reading and writing.

w+ Same as w except both for reading and writing.

a+ Same as a except both for reading and writing.

Closing A File

A file must be closed as soon as all operations on it have' been completed. This ensures that all outstanding information associated with the file is flushed out from the buffers and all links to the file are broken. It also prevents any accidental misuse of the file. The I/O library supports a function to do this . It takes the following form:

fclose(file_pointer);

This would close the file associated with the FILE pointer file_pointer. Look at the following segment of a program.

FILE *p1, *p2;

p1 = fopen("INPUT", "w");

p2 = fopen("OUTPUT", "r");

fclose(p1); fclose(p2);

This program opens two files and closes them after all operations on them are completed. Once a file is closed, its file pointer can be reused *for* another file.

Input/Output Operations On Files

Once a file is opened, reading out of or writing to it is accomplished using the standard routines that are listed in Table 1.

The getc and putc Functions

The simplest file *I/O* functions are getc and pute. These are analogous to getehar and putchar functions and handle one character at a time. Assume that a file is opened with mode w and file pointer fp1. Then, the statement

```
putc(c, fp1);
```

writes the, character contained in the character variable c to the file associated with FILE pointer fpl. Similarly, getc is used to read a character from a file that has been opened in read mode. For example, the statement

```
c = getc(fp2);
```

would read a character from the file whose file pointer is fp2.

The file pointer moves by one character position *for* every operation of getc(). The getc() will return an end-of-file marker EOF, when end of the file has been reached. Therefore, the reading should be terminated when EOF is encountered.

Example Program 1

Write a program to read data from the keyboard, write it to a file called INPUT, again read the same data *from* the INPUT file, and display it on the screen.

```
 #include <stdio.h>
main( )
{
FILE *f1;
 char c;
printf("Data Input\ n\ n");
f1 = fopen("INPUT", “w");        /* Open the file INPUT */
while((c=getchar()) != EOF)    /* Get a character from keyboard*/
putc(c,f1);                        /* Write a character to INPUT file */
fclose(f1);
printf("\nData Output\n\n");
f1 = fopen("'NPUT","r");                /*.Reopen the file INPUT */
while((c=getc(f1) != EOF)     /* Read a character from INPUT*/
printf("%c",c);
fclose(f1);                /* Close the file INPUT


}
```

The getw and putw Functions

The getw and putw are integer-oriented functions. They ,are similar to the getc and putc functions and are used to read and write integer values. These functions would be useful when we deal with only integer data. The general forms of getw and putw are:

putw(integer_variable ,fp);

integer_variable =getw(fp);

Example .2

A file named DATA contains a series of integer numbers. Code a program to read these numbers and then write all odd numbers to a file to be called ODD and all even numbers to a file to be called EVEN.

```
#include <stdio.h>
main( )
{
FILE *f1, *f2, *f3;
int number, i;
printf(“Contents of DATA file(Input -1 to terminate)\n\n”);
f1 =fopen(“DATA”, “w”);
while(1)
{
scanf(U%d", &number);
if(number = = -1)
break;
putw(number,f1 );
}
fclose(f1 );
f1 = fopen(“DATA”, “r");
f2 = fopen(“ODD", “w");
f3 = fopen(“EVEN", “w");
while((number = getw(f1)).! = EOF)
 {
  if(number %2 == 0)
  putw(number, f3);
  else
   putw(number, f2);
  fclose(f1 );
  fclose(f2);
  fclose(f3);
  f2 = fopen(“ODD",”r");
  f3 = fopen(“EVEN", “r");
  printf(“\n\nContents of ODD file\n\n");
  while((number = getw(f2)) ! = EOF)
   printf("%4d", number);
   printf("\n\nContents of EVEN file\n\n");
   while((number = getw(f3)) ! = EOF)
   printfr%4d" number);
  fclose(f2);
  fClose(f3);
}
```

<u>The fprintf and fscanf Functions</u>

So far, we have seen functions which can handle only one character or integer at a time. Most compilers support two other functions, namely fprintf and scanf, that can handle a group of mixed data simultaneously.

The functions fprintf and fscanf perform I/O operations that are identical to the familar printf and scanf functions, except of course that they work on files. The first argument of these functions is a file pointer which specifies the file to be used. The general form of **fprintf** is

```
fprintf(fp, “control string” ,list)
```

Where fp is pointer associated with a file that has been opened for writing. The control string contains output specifications for variables specified in the list. The list may contain variables , constants and strings.

For example

```
fprintf(f1, "%S %d %f", name, age, 7.5);
```

Here, name is an array variable of type char and age is an int variable.

The general format of fscanf is

fscanf(fp, *"control string", list);*

This statement would cause the reading of the items in the *list* from the file specified by *fp,* according to the specifications contained in the *control string.* Example:

```
fscanf(f2, "%s %d", Item, &quantlty);
```

Like scanf, fscanf also returns the number of items that are successfully read. When the end of file is reached, it returns the value EOF.

Example 3

Write a program to open a file named INVENTORY and store in it the following data:

Item name *Number* *Price* *Quantity*

Extend the program to read this data from the file INVENTORY and display the inventory table with the value of each item.

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

FILE *fp;
int number, quantity, i;
float price, value;
char item[10], filename[10];
printf("lnput file name\n");
scanf("%s", filename);
fp = fopen(filename, "w");
printf("lnput inventory data\n\n");
printf("ltem name NumberPrice  Quantity\n");
for(i = 1; i <= 3; i++)
{
fscanf(stdin, "%s %d %f %d",
                item, &number, &price, &quantity);
fprintf(fp, "%s %d %.2f %d",
                item, number, price, quantity);
}
fclose(fp);
fprintf(stdout, "\n\ n");
fp = fopen(filename. "r");
printf("ltem name Number        Price       Quantity  Value\n");
for(i = 1; i <= 3; i++)
{
fscanf(fp, "%s %d %f %d",item,&number,&price,&quantity); value = price' quantity;
fprintf(stdout, "% -8s % 7d %8.2f %8d % 11.2f\n",
item, number. price, quantity, value);
}
 fclose(fp );
}
```

****