

**GOVERNMENT OF TAMILNADU
DIRECTORATE OF TECHNICAL EDUCATION
CHENNAI – 600 025
STATE PROJECT COORDINATION UNIT**

Diploma in Computer Engineering

Course Code: 1052

**M – Scheme
e-TEXTBOOK
on
C PROGRAMMING
for
III Semester Diploma in Computer Engineering**

Convener for Computer Engineering Discipline:

Mr. D. Arulselvan,
HOD /PDCA,
Thiagarajar Polytechnic College ,
Salem - 636 005.

Team Members for C Programming:

D.Arulselvan
HoD / PDCA
Thiagarajar Polytechnic College,
Salem – 636 005.

V. Saranya
HoD / Computer Engineering
Thiagarajar Polytechnic College,
Salem – 636 005.

M.Gomathi,
Lecturer/ Computer Engineering
Thiagarajar Polytechnic College,
Salem – 636 005.

Validated by

Mr. B.Krishnakumar
HOD / Computer Engineering
Arasan Ganesan Polytechnic College
Sivakasi

STATE BOARD OF TECHNICAL EDUCATION & TRAINING, TAMILNADU
DIPLOMA IN COMPUTER ENGINEERING
M- SCHEME

Course Name : COMPUTER ENGINEERING

Subject Code : 35233

Semester : III

Subject Title : C PROGRAMMING

TEACHING & SCHEME OF EXAMINATION

No. of weeks per semester: 15 weeks

Subject	Instructions		Examination			Duration
			Max. Marks			
	Hours / week	Hours / Semester	Internal Assessment	Board Examinations	Total	
Data Structures Using C	5	75	25	75	100	3 Hours

TOPICS AND ALLOCATION OF HOURS

Unit No	Topic	No of Hours
I	PROGRAM DEVELOPMENT AND INTRODUCTION TO C	12
II	DECISION MAKING, ARRAYS AND STRINGS	13
III	FUNCTIONS, STRUCTURES AND UNIONS	13
IV	POINTERS	14
V	FILE MANAGEMENT & PREPROCESSORS	13
CONTINUOUS ASSESSMENT TEST AND MODEL EXAMS		10
TOTAL		75

RATIONALE

C' is the most widely used computer language, which is being taught as a core course. C is general purpose structural language that is powerful, efficient and compact, which combines features of high level language and low-level language. It is closer to both Man and Machine. Due to this inherent flexibility and tolerance it is suitable for different development environments. Due to these powerful features, C has not lost its importance and popularity in recently developed and advanced software industry. C can also be used for system level programming and it is still considered as first priority programming language. This course covers the basic concepts of C. This course will act as "Programmingconcept developer" for students. It will also act as "Backbone" for subjects like OOPS, Visual Basic, Windows Programming, JAVA etc.

OBJECTIVES

At the end of the Course, the students will be able to

- Define Program, Algorithm and flow chart
- List down and Explain various program development steps
- Write down algorithm and flow chart for simple problems.
- Describe the concepts of Constants, Variables, Data types and operators. Develop programs using input and output operations.
- Use of command line arguments & Explain compiler controlled directives.
- Understand the structure and usage of different looping and branching statements. Define arrays and string handling functions.
- Explain user-defined functions, structures and union.
- Define pointers and using the concept of Pointers.
- To understand the dynamic data structure and memory management.

DETAILED SYLLABUS

UNIT I PROGRAM DEVELOPMENT & INTRODUCTION TO C	 12 HOURS
1.1	Program Algorithm & flow chart: Program development cycle- Programming language levels & features. Algorithm – Properties & classification of Algorithm, flow chart – symbols, importance & advantage of flow chart.	2 Hrs
1.2	Introduction C: - History of C – features of C structure of C program –Compiling, link & run a program. Diagrammatic representation of program execution process.	2 Hrs
1.3.	Variables, Constants & Data types: C character set-Tokens- Constants- Key words – identifiers and Variables – Data types and storage – Data type Qualifiers – Declaration of Variables – Assigning values to variable - Declaring variables as constants-Declaration – Variables as volatile- Overflow & under flow of data	3 Hrs
1.4	C Operators: Arithmetic, Logical, Assignment Relational, Increment and Decrement, Conditional, Bitwise, Special Operator precedence and Associativity. C expressions – Arithmetic expressions – Evaluation of expressions- Type cast operator	3 Hrs
1.5	I/O statements: Formatted input, formatted output, Unformatted I/O statements	2 Hrs
UNIT II DECISION MAKING,ARRAYS and STRINGS	 13 HOURS
2.1.	Branching: Introduction – Simple if statement – if –else – else-if ladder , nested if-else- Switch statement – go statement – Simple programs.	4 Hrs
2.2.	Looping statements: While, do-while statements, for loop, break & continue statement – Simple programs	3 Hrs
2.3.	Arrays: Declaration and initialization of One dimensional, Two dimensional and character arrays – Accessing array elements – Programs using arrays	3 Hrs
2.4	Strings : Declaration and initialization of string variables, Reading String, Writing Strings – String handling functions (strlen(),strcat(),strcmp()) – String manipulation programs	3 Hrs

UNIT III FUNCTIONS, STRUCTURES AND UNIONS		 13 HOURS
3.1.	Built -in functions: Math functions – Console I/O functions – Standard I/O functions – Character Oriented functions – Simple programs		3 Hrs
3.2	User defined functions: Defining functions & Needs-, Scope and Life time of Variables, , Function call, return values, Storage classes, Category of function – Recursion – Simple programs		6 Hrs
3.3	Structures and Unions: Structure – Definition, initialization, arrays of structures, Arrays with in structures, structures within structures, Structures and functions – Unions – Structure of Union – Difference between Union and structure – Simple programs.		4 Hrs
UNIT IV POINTERS		 14 HOURS
4.1.	Pointers: Definition – advantages of pointers – accessing the address of a variable through pointers - declaring and initializing pointers- pointers expressions, increment and scale factor- array of pointers- pointers and array - pointer and character strings – function arguments – pointers to functions – pointers and structures – programs using pointer.		10 Hrs
4.2.	Dynamic Memory Management: introduction – dynamic memory allocation – allocating a block memory (MALLOC) – allocating multiple blocks of memory (CALLOC) –releasing the used space: free – altering the size of a block (REALLOC) – simple programs		4 Hrs
UNIT V FILE MANAGEMENT AND PREPROCESSORS		 13 HOURS
5.1	File Management: Introduction-Defining and opening a file-closing a file-Input/ Output operations on files—Error handling during I/O operations –Random Access to files— Programs using files		8 Hrs
5.2	Command line arguments: Introduction – argv and argc arguments – Programs using command Line Arguments –Programs		2 Hrs
5.3	The preprocessor: Introduction – Macro Substitution, File inclusion, Compiler control directives.		3 Hrs

TEXT BOOKS

S.No	TITLE	AUTHOR	PUBLISHER	YEAR OF PUBLISHING / EDITION
1.	Programming in ANSI C	Prof. E. BALAGURUSAMY	TATA McGRAWHILL publications.	4th Edition

REFERENCES

S.No	TITLE	AUTHOR	PUBLISHER	YEAR OF PUBLISHING/EDIT ION
1.	Programming and Problem solving using C	ISRD Group, Lucknow	Tata Mc-GrawHill, NewDelhi	Sixth Reprint 2010
2.	Let us C	Yeswanth Kanetkar	BPB Publications	Fourth Revised
3.	A TextBook on C	E.Karthikeyan	PHI Private Limited, New Delhi	2008

4.	Programming in C	D.Ravichandran	New Age International Publishers,Chennai	FirstEdition1996 Reprint2011
5.	Computer Concepts and	Dr.S.S.Khandare	S.Chand & Company Ltd. New Delhi	FirstEdition2010
6.	Complete Knowledge in C	Sukhendu Dey, Debabrata Dutta	Narosa Publishing House, New Delhi	Reprint2010
7.	Programming in C	Reema Theraja	Oxford University Press	FirstEdition2011
8.	Practical C Programming	Steve Oualline	O'Reilly, Shroff	Eleventh Indian ReprintOct2010

CONTENTS

Unit No	Name of the Unit	Page No
I	PROGRAM DEVELOPMENT & INTRODUCTION TO C	01
II	DECISION MAKING, ARRAYS and STRINGS	48
III	FUNCTIONS, STRUCTURES AND UNIONS	96
IV	POINTERS	134
V	MANAGEMENT AND PREPROCESSORS	165

PROGRAM DEVELOPMENT & INTRODUCTION TO C

OBJECTIVES

At the end of the unit, the students will be able to

- Explain program Development Life Cycle
- Understand algorithm and its properties
- Understand flow chart and its uses
- Write algorithms for simple programs
- Draw flow chart for simple programs
- Understand the basic structure of C
- Obtain a preliminary idea of the keywords in C
- Learn the data types, variables, constants, operators and expressions in C
- Get acquainted with the rules of types of expressions in C.
- Learn input / output functions.
- Differentiate formatted and unformatted I/O functions.

INTRODUCTION

Before any problem can be solved using a computer, the person writing the program must be familiarized with the problem and with the way in which it has to be solved. The problem solved is to be represented by small and clear steps by using algorithms and flow charts.

A program is a set of instructions to solve a particular problem. C is a programming language , is to be considered as a middle level language . This unit gives an overview of C language and explains about the structure of C program. Variables and constants joined by various operators form an expression, Basic I/O functions are used to accept data and produces output. Different types of values may be passed to the computer from the keyboard. Such different types for the values are called as data types.

This unit will discuss in detail about the data types, variables, constants, various categories of operators , expressions, type modifiers and Input – Output operations

1.1 PROGRAM , ALGORITHM & FLOW CHART

1.1.1 PROGRAM - DEFINITION

A computer program is a sequence of instructions written to perform a specified task with a computer.

Programs are written in a programming language. These programs are then translated into machine code by a compiler and linker so that the computer can execute it directly or run it line by line (interpreted) by an interpreter program.

1.1.2 PROGRAM DEVELOPMENT LIFE CYCLE

The process of developing software, according to the desired needs of a user, by following a basic set of interrelated procedures is known as Program Development Life Cycle (PDLC)

PDLC includes various set of procedures and activities that are isolated and sequenced for learning purposes but in real life they overlap and are highly interrelated.

Tasks of Program Development

The basic set of procedures that are followed by various organizations in their program development methods are as follows:

1. Program specification.
2. Program Design.
3. Program coding.
4. Program testing.
5. Program documentation.
6. Program Maintenance.



Fig No 1.1. Program Development Life Cycle

1. PROGRAM SPECIFICATION

This stage is the formal definition of the task. It includes the specification of inputs and outputs, processing requirements, system constraints, and error handling methods.

This step is very critical for the completion of a satisfactory program. It is impossible to solve a problem by using a computer, without a clear understanding and identification of the problem. Inadequate identification of problem leads to poor performance of the system. The programmer should invest a significant portion of his time in problem identification.

2. PROGRAM DESIGN

In this phase the design of the system is designed. The Design is developed by the analysts and designers. The system analyst design the logical design for the designers and then designer get the basic idea of designing the software design of Front end and back end both.

The system analyst and Designer work together in designing the software design and designer design the best software design under the guidance of system analyst.

3. PROGRAM CODING

This step transforms the program logic design documents into a computer language format. This stage translates program design into computer instructions. These instructions are the actual program. It is the crucial job of programmer to develop a code by following the flowchart and that code is written in any computer language.

4. PROGRAM TESTING

Program testing is the process of checking program, to verify that it satisfies its requirements and to detect errors. These errors can be of any type - Syntax errors, Run-time errors and Logical errors . Testing include necessary steps to detect all possible errors in the program. This can be done either at a module level known as unit testing or at program level known as integration testing.

Syntax errors also known as compilation errors are caused by violation of the grammar rules of the language. The compiler detects, isolate these errors and terminate the source program after listing the errors. Common syntax errors include

- missing or misplaced; or },
- missing return type for a procedure,
- Missing or duplicate variable declaration.
- Type errors that include
 - type mismatch on assignment,
 - type mismatch between actual and formal parameters.

Logical errors: These are the errors related with the logic of the program execution. These errors are not detected by the compiler and are primarily due to a poor understanding of the problem or a lack of clarity of hierarchy of operators. Such errors cause incorrect result.

Errors such as mismatch of data types or array out of bound error are known as execution errors or **runtime errors**. These errors are generally undetected by the compiler, so programs with run-time error will run but produce erroneous results.

Debugging is a methodical process of finding and reducing the number of bugs in a computer program making it behave as expected.

5. PROGRAM DOCUMENTATION

This task is performed by the programmer to make the code user friendly i.e. if new person got the code then he/she can easily understand that which statement performed what task.

Proper documentation is useful in the testing and debugging stages. It is also essential in the maintenance and redesign stages. A properly documented program can be easily reused when needed; an undocumented program usually requires so much extra work that the programmer might as well start from scratch. The techniques commonly used in documentation are flowcharts, comments, memory maps, parameter and definition lists, and program library forms.

Proper documentation combines all or most of the methods mentioned. Documentation is a time-consuming task. The programmer performs this task simultaneously with the design, coding, debugging and testing stages of software development. Good documentation simplifies maintenance and redesign, and makes subsequent tasks simpler.

6. PROGRAM MAINTENANCE

During this phase, the program is actively used by the users. If the user encounters any problem or wants any enhancement, then repeat all the phases from the starting, so that the encountered problem is solved or enhancement is added.

1.1.3 PROGRAMMING LANGUAGES AND FEATURES

Programming language is a set of grammatical rules for instructing a [computer](#) to perform specific tasks. The term *programming language* usually refers to [high-level languages](#), such as [BASIC](#), [C](#), [C++](#), [COBOL](#), [FORTRAN](#), [Ada](#), and [Pascal](#). There are two major types of programming languages. These are Low Level Languages and High Level Languages. Low Level languages are further divided in to *Machine language* and *Assembly language*.

LOW LEVEL LANGUAGES

Low level languages are machine oriented and require knowledge of computer hardware and its configuration.

(a) Machine Language

Machine Language is the only language that is directly understood by the computer. It does not need any translator program. It is written as strings of 1's (one) and 0's (zero). For example, a program instruction may look like this:

1011000111101

It is not an easy language to learn because of its difficult to understand. It is efficient for the computer but very inefficient for programmers. It is considered to the first generation language. It is also difficult to debug the program written in this language.

Advantage : Machine language programs are run very fast because no translation program is required for the CPU.

Disadvantages

1. It is very difficult to program in machine language. The programmer should know details of hardware to write program.
2. The programmer should remember a lot of codes to write a program which results in program errors.
3. It is difficult to debug the program.

(b) Assembly Language

The computer can handle numbers and letter. Therefore, some combination of letters can be used to substitute for number of machine codes. The set of symbols and letters forms the Assembly Language and a translator program is required to translate the assembly Language to machine language. This translator program is called 'Assembler'.

Advantages:

1. Assembly Language is easier to understand and saves a lot of time and effort of the programmer.
2. It is easier to correct errors and modify program instructions.
3. Assembly Language has the same efficiency of execution as the machine level language. Because this is one-to-one translator between assembly language program and its corresponding machine language program.

Disadvantages:

1. The assembly language is machine dependent. A program written for one computer might not run in other computers with different hardware configuration.

HIGH LEVEL LANGUAGES

The assembly language and machine level language require deep knowledge of computer hardware. But High-level languages are machine independent. Programs are written in English-like statements. As high – level languages are not directly executable, translators (compilers or interpreters) are used to convert them into machine language.

Advantages of High Level Languages

1. These are easier to learn. Less time is required to write programs.
2. They are easier to maintain. Programs written in high-level languages are easier to debug.
3. Programs written in high-level languages are machine –dependent. Therefore programs developed on one computer can be run on another with little or no modifications.

1.1.4 FEATURES OF A GOOD PROGRAMMING LANGUAGES

Simplicity: A good programming language must be simple and easy to learn and use. For example, BASIC is liked by many programmers only because of its simplicity. Thus, a good programming language should provide a programmer with a clear, simple and unified set of concepts which can be easily grasped.

Naturalness: A good language should be natural for the application area it has been designed. That is, it should provide appropriate operators, data structures, control structures, and a natural syntax in order to facilitate the users to code their problem easily and efficiently.

Abstraction: Abstraction means the ability to define and then use complicated structures. The degree of abstraction allowed by a programming language directly affects its writability.

Efficiency: The programs written in good programming languages are efficiently translated into machine code, are efficiently executed, and acquire as little space in the memory as possible.

Compactness: In a good programming language, programmers should be able to express intended operation concisely.

Locality: A good programming language should be such that while writing a program, a programmer need not jump around visually as the text of the program is prepared. This allows the programmer to concentrate almost solely on the part of the program around the statements currently being worked with.

Extensibility: A good programming language should allow extension through simple, natural, and elegant mechanisms. Almost all languages provide subprogram definition mechanisms for this purpose.

1.1.5 ALGORITHM - DEFINITION

1.1.5.1. ALGORITHM - DEFINITION

Algorithm is a step-by-step method of solving a problem or making decisions.

1.1.6 PROPERTIES OF ALGORITHM

1. **Finiteness:** An algorithm must always terminate after a finite number of steps. It means after every step one reaches closer to solution of the problem and after a finite number of steps algorithm reaches to an end point.
2. **Definiteness:** Each step of an algorithm must be precisely defined. It is done by well thought actions to be performed at each step of the algorithm. Also the actions are defined unambiguously for each activity in the algorithm.
3. **Input:** Any operation to be performed needs some beginning value/quantities associated with different activities in the operation. So the value/quantities are given to the algorithm before it begins.
4. **Output:** One always expects output/result (expected value/quantities) in terms of output from an algorithm. The result may be obtained at different stages of the algorithm. If some result is from the intermediate stage of the operation then it is known as intermediate result and result obtained

at the end of algorithm is known as end result. The output expected value/quantities always have a specified relation to the inputs.

5. Effectiveness: Algorithms to be developed/written using basic operations.

Any algorithm should have all these five properties otherwise it will not fulfill the basic objective of solving a problem in finite time.

1.1.7 CLASSIFICATION OF ALGORITHMS

An algorithm may be implemented according to different basic principles.

- **Recursive or iterative:** A recursive algorithm is one that calls itself repeatedly until a certain condition matches. It is a method common to functional programming. Iterative algorithms use repetitive constructs like loops.
- **Logical or procedural:** An algorithm may be viewed as controlled logical deduction. A logic component expresses the axioms which may be used in the computation and a control component determines the way in which deduction is applied to the axioms.
- **Serial or parallel:** Algorithms are usually discussed with the assumption that computers execute one instruction of an algorithm at a time. This is a serial algorithm, as opposed to parallel algorithms, which take advantage of computer architectures to process several instructions at once. They divide the problem into sub-problems and pass them to several processors.
- **Deterministic or non-deterministic:** Deterministic algorithms solve the problem with a predefined process whereas non-deterministic algorithm must perform guesses of best solution at each step through the use of heuristics.

1.1.8 ALGORITHMS LOGIC

Algorithmic logic is classified into three types. They are: (i) Sequential Logic (ii) Selection or Conditional Logic and (iii) Repetition logic. All the computing is done using only these types.

SEQUENTIAL LOGIC

As the name suggests, Sequential Logic consist of one action followed by another in a logical progression. In other words, perform operation A and then perform operation B and so on. This structure is represented by writing one operation after another.

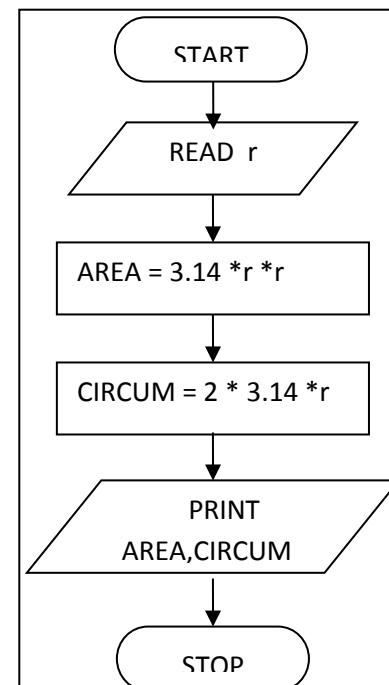


Fig No 1.2 Example for Sequential Logic

SELECTION OR CONDITIONAL LOGIC

Selection Logic allows the program to make a choice between two alternate paths, whether it is true (1) or false (0). First statement is a conditional statement.

If the condition is true, Operation 1 is performed; otherwise Operation 2 will be performed. For example, if $A > B$, then print A, else print B

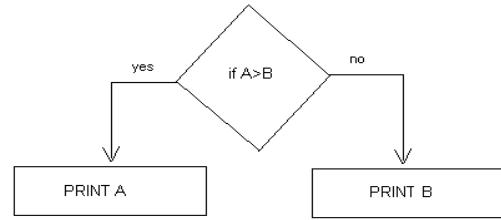


Fig No 1.3 Example for Selection or Conditional Logic

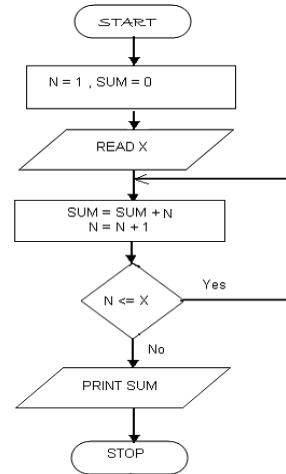
This is called the “if...Then...Else” structure. If the answer is “Yes”, then the control will be transferred to some other path. Otherwise, if the answer is “No”, then the execution goes to the next statement without doing anything.

ITERATION OR REPETITION LOGIC

Repetition Control Structure is also termed as Iteration Control Structure or Program Loop. Repetition causes an interruption in the sequence of processing. In the Repetitive Control Structure, an operation or a set of operations is repeated as long as some condition is satisfied. The performed operation will be the same, but the data will change every time.

Fig No 1.4 Example for Repetition Logic

When a sequence of statements is repeated against a condition, it is said to be in a loop. Using looping, the programmer avoids writing the same set of instructions again and again.



Some examples on developing algorithms using step-form:

- Each algorithm will be logically enclosed by two statements START and STOP.
- Input or READ statements are used to accept data from user.
- PRINT statement is used to display any user message. The message will be displayed to be enclosed within quotes.
- The arithmetic operators $=, +, *, -, /$ will be used in the expression.
- The commonly used relational operators will include $:>, <, >=, <=, +, !=$
- The most commonly used logical operators will be AND, OR, NOT

Example 1 : To make a coffee

Step1: Take proper quantity of water in a cooking pan

Step2: Place the pan on a gas stove and light it

Step3: Add Coffee powder when it boils

Step4: Put out the light and add sufficient quantity of sugar and milk

Step5: Pour into cup and have it.

Example 2 : To find the area and circumference of a circle

1. Start
2. Read the radius of the circle r
3. Find the area and circumference of the circle using formula
Area = $3.14 \times r^2$
Circum = $2 \times 3.14 \times r$
4. Print the area and circumference of the circle
5. Stop

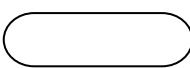
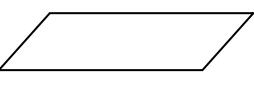
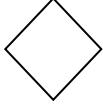
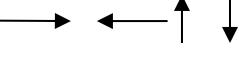
1.1.9 FLOW CHART

A flowchart is a diagrammatic representation that shows the sequence of operations to be performed to get the solution of a problem.

IMPORTANCE OF FLOW CHART

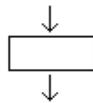
Flowcharts facilitate communication between programmers and business people. Flowcharts play a vital role in the programming of a problem and are quite helpful in understanding the logic of complicated and lengthy problems. Once the flowchart is drawn, it becomes easy to write the program in any high level language. Flowcharts are helpful in explaining the program to others. Hence, a flowchart is a must for the better documentation of a complex program.

FLOWCHART SYMBOLS

S.No	Name of the Symbol	Symbol	Meaning
1	Start /Stop (Oval)		Represents the start and end of the program
2	Processing (Rectangle)		Represents arithmetic and data movement instructions.
3	Input / Output (Parallelogram)		It is used to represent the input or output function.
4.	Decisions (Diamond)		Represents decision making and branching. It has one entry and two or more exit paths
5.	Connector (Circle)		Used to join two parts of a program
6	Flow lines (Arrow lines)		Used to link various boxes together to form the flow chart and indicate the direction of the flow.

GUIDELINES FOR DRAWING FLOWCHART

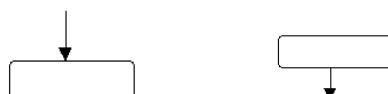
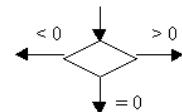
- First of all list all necessary requirements in a logical order.
- The flowchart should be clear and easy to understand. There should not be any ambiguity in understanding the flowchart.
- The usual direction of the flow of a procedure or system is from left to right or top to bottom.
- Only one flow line should come out from a process symbol.



or



- Only one flow line should enter a decision symbol, but two or three flow lines, one for each possible answer, should leave the decision symbol.
- Only one flow line is used in conjunction with terminal symbol.



- If the flowchart becomes complex, it is better to use connector symbols to reduce the number of flow lines. Avoid the intersection of flow lines for more effective and better way of communication.
- Ensure that the flowchart has a logical *start* and *finish*.
- It is useful to test the validity of the flowchart by passing through it with a simple test data.

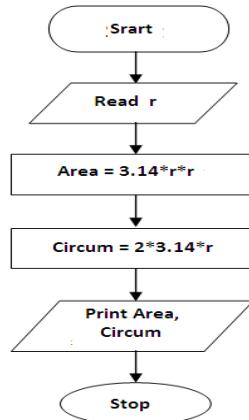
ADVANTAGES OF USING FLOWCHARTS

1. **Communication:** Flowcharts are better way of communicating the logic of a system to all concerned.
2. **Effective analysis:** With the help of flowchart, problem can be analysed in more effective way.
3. **Proper documentation:** Program flowcharts serve as a good program documentation, which is needed for various purposes.
4. **Efficient Coding:** The flowcharts act as a guide or blueprint during the systems analysis and program development phase.
5. **Proper Debugging:** The flowchart helps in debugging process.
6. **Efficient Program Maintenance:** The maintenance of operating program becomes easy with the help of flowchart. It helps the programmer to put efforts more efficiently on that part

LIMITATIONS OF USING FLOWCHARTS

1. **Complex logic:** Sometimes, the program logic is quite complicated. In that case, flowchart becomes complex and clumsy.
2. **Alterations and Modifications:** If alterations are required the flowchart may require re-drawing completely.

FLOW CHART FOR FINDING THE AREA AND CIRCUMFERENCE OF CIRCLE

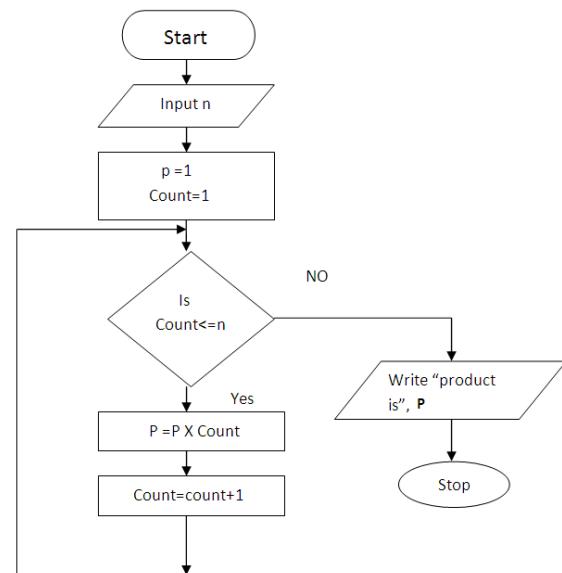


Algorithm and flow chart for finding the product of first n natural numbers

ALGORITHM

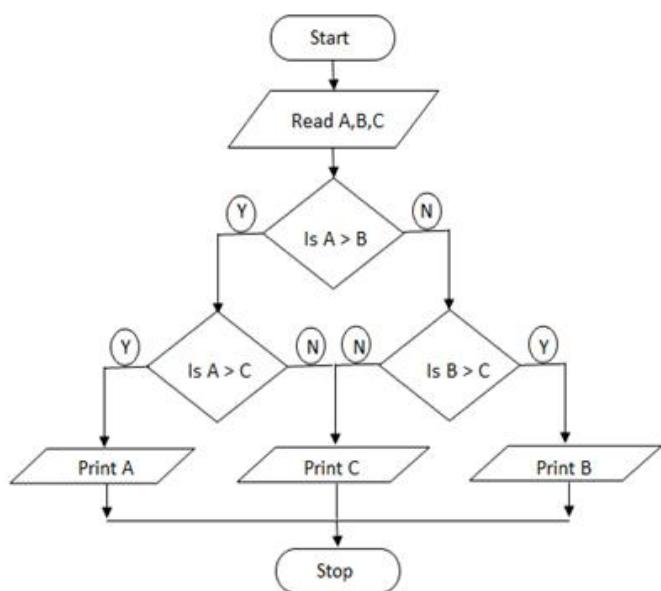
1. Start
2. Read the number n.
3. Set the values P $\leftarrow 1$, Count $\leftarrow 1$
4. While (count $\leq n$)
 - do
 - P $\leftarrow P \times$ count
 - count \leftarrow count +1
 - End the while loop.
5. Write "product of num is", P.
6. Stop

FLOW CHART



Algorithm and flow chart for finding the largest of 3 numbers

1. Start.
2. Read three numbers s A, B and C.
3. Find A is greater than B and C, if so print the variable A.
4. If B is greater than A, find B is greater than C, if so print the variable B.
Else print the variable C.
5. Stop

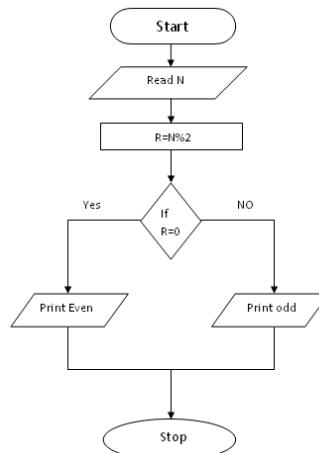


Algorithm and flow chart for finding whether the given number is odd or even.

ALGORITHM

1. Start
2. Read the number N.
3. Find the remainder(R) of N divided by 2 using the modulus operator(N%2)
4. If the remainder is zero.
Print "The number is Even Number".
Else
Print "The number is odd number".
5. Stop.

FLOW CHART



1.2 INTRODUCTION TO C

1.2.1 HISTORY OF C

C is a general purpose computer programming language and was developed at AT&T's bell laboratory of USA in 1972. It was originally created by Dennis Ritchie. By 1960, different computer languages were used for different purposes. So, an International Committee was set up to develop a language that is suitable for all purposes. This language is called ALGOL 60.

To overcome the limitation of ALGOL 60, a new language called *Combined Programming Language (CPL)* was developed at Cambridge University. CPL has so many features. But it was difficult to learn and implement. Martin Richards of Cambridge University developed a language called "*Basic Combined Programming Language*" (*BCPL*). BCPL solves the problem of CPL. But it is less powerful. At the same time, Ken Thomson at AT&T's Bell laboratory developed a language called 'B'. Like BCPL, B is also very specific. Ritchie eliminated the limitations of B and BCPL and developed 'C'.

1960	ALGOL	International Group	1972	Traditional C	Dennis Ritchie
1964	CPL	Cambridge University	1978	K&R C	Kernighan & Ritchie
1967	BCPL	Martin Richards	1989	ANSI C	ANSI Committee
1970	B	Ken Thompson	1990	ANSI/ISO C	ISO Committee

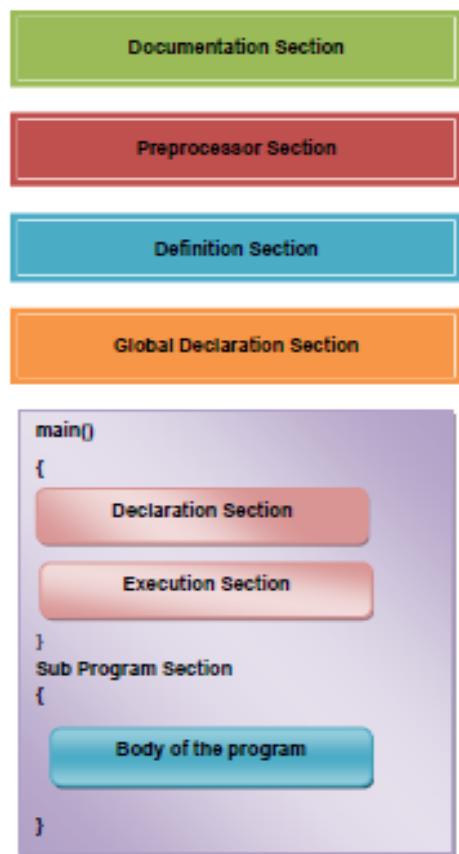
1.2.2 FEATURES OF C LANGUAGE

- C has all the advantages of assembly language and all the significant features of modern high-level language. So it is called a "*Middle Level Language*".
- C language is a very powerful and flexible language.
- C language supports a number of data types and consists of rich set of operators.
- C language provides dynamic storage allocation.
- C Compiler produces very fast object code.
- C language is a portable language. A code written in C on a particular machine can be compiled and run on another machine.

1.2.3 STRUCTURE OF A C PROGRAM

Any C Program consists of one or more function. A function is a collection of statements, used together to perform a particular task. An overview of the structure of a C program is given below: A C program may contain one or more sections. They are illustrated below.

Documentation Section: The documentation section consists of a set of comment lines giving the name of the program, the author and other details. It consist of a set of comment lines . These lines are not executable. Comments are very helpful in identifying the program features



Preprocessor Section: It is used to link system library files, for defining the macros and for defining the conditional inclusion.

Definition section : The definition section defines all symbolic constants.

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

main () function section : Every C program must have one main function section. This section contains two parts; declaration part and executable part.

- a) Declaration part : The declaration part declares all the variables used in the executable part.
 - b) Executable part : There is at least one statement in the executable part.

These two parts must appear between the opening and closing braces. The program execution begins at the opening brace and ends at the closing brace.

Fig No 1.5 Structure of C Program

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

SAMPLE C PROGRAM

```
#include <stdio.h>

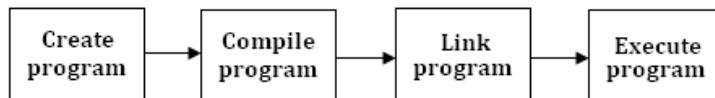
int main () { int number, remainder;
printf ("Enter your number to be tested: ");
scanf ("%d", &number);
remainder = number % 7;
if ( remainder == 0 )
printf ("The number is divided by 7.\n");
else
printf ("The number is not divided by 7.\n"); }
```

General rule for C Programming:

1. Every executable statement must end with semicolon symbol (;) .
2. Every C Program, must contain exactly one main method (starting point of the program execution)
3. All the system defined words (keywords) must be used in lowercase letters.
4. Keywords cannot be used as user defined names.
5. For every open brace ({}, there must be respective closing brace}).

1.2.4 EXECUTING A “C” PROGRAM

C program executes in following four steps.: 1. Creating the program 2. Compiling the Program
3. Linking the Program with system library 4. Executing the program



1. **Creating a program:** Type the program and edit it in standard ‘C’ editor and save the program with .c as an extension. This is the source program .The file should be saved as ‘*.c’ extension only.
2. **Compiling (Alt + F9) the Program:**
 - This is the process of converting the high level language program to Machine level Language (Equivalent machine instruction) .
 - Errors will be reported if there is any, after the compilation
 - Otherwise the program will be converted into an object file (.obj file) as a result of the compilation
 - After error correction the program has to be compiled again
3. **Linking a program to library:** The object code of a program is linked with libraries that are needed for execution of a program. The linker is used to link the program with libraries. It creates a file with ‘*.exe’ extension.
4. **Execution of program:** This is the process of running (Ctrl + F9) and testing the program with sample data. If there are any run time errors, then they will be reported.

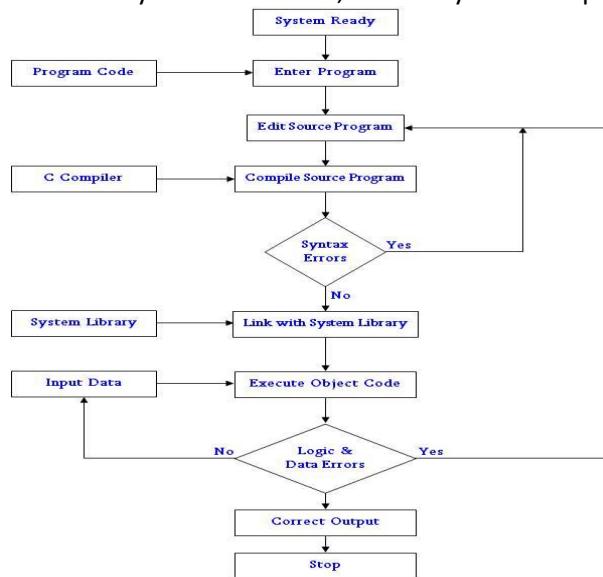


Figure.: Process of compiling and running a C program.

1.3 VARIABLES , CONSTANTS AND DATA TYPES

1.3.1. C CHARACTER SET

Every C program contains statements. These statements are constructed using words and these words are constructed using characters from C character set. C language character set contains the following set of characters...

1. Alphabets
2. Digits
3. Special Symbols

Alphabets: C language supports all the alphabets from english language. Lower and upper case letters together supports 52 alphabets.

lower case letters - **a to z**

UPPER CASE LETTERS - **A to Z**

Digits : C language supports 10 digits which are used to construct numerical values in C language.

Digits - **0, 1, 2, 3, 4, 5, 6, 7, 8, 9**

Special Symbols : C language supports rich set of special symbols that include symbols to perform mathematical operations, to check conditions, white spaces, back spaces and other special symbols.

,	Comma	&	Ampersand
.	Period	^	Caret
;	Semicolon	*	Asterisk
:	Colon	-	Minus Sign
?	Question Mark	+	Plus Sign
'	Apostrophe	<	Opening Angle (Less than sign)
"	Quotation Marks	>	Closing Angle (Greater than sign)
!	Exclamation Mark	(Left Parenthesis
	Vertical Bar)	Right Parenthesis
/	Slash	[Left Bracket
\	Backslash]	Right Bracket
~	Tilde	{	Left Brace
-	Underscore	}	Right Brace
\$	Dollar Sign	#	Number Sign
%	Percentage Sign		

White Space Characters:

The compiler ignores white spaces unless they are a part of a string constant. White spaces may be used to separate words in strings. scanf() uses whitespace to separate consecutive input items from each other.

1. Blank Space
2. Horizontal Tab
3. Carriage Return
4. New Line
5. Form Feed

1.3.2. C TOKENS

C tokens are the basic buildings blocks in C. Smallest individual units in a C program are the C tokens. C tokens are of six types. The figure 1.7 shows the C tokens.

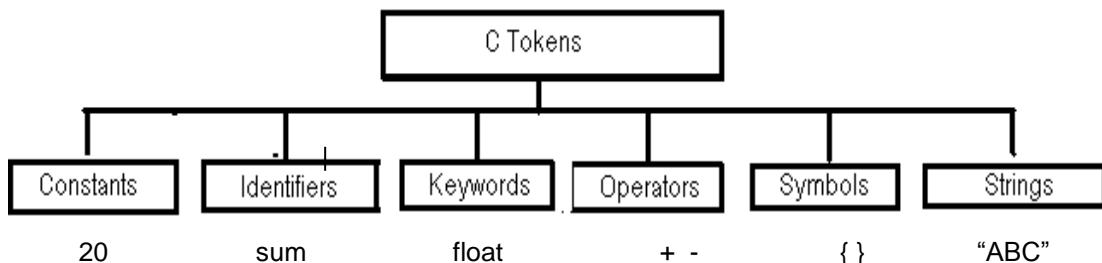


Fig No 1.7 C Tokens

1.3.3. KEYWORDS

The meaning of these words has already been explained to the C compiler. All the keywords have fixed meanings. These meanings cannot be changed. So these words cannot be used for other purposes. *All keywords are in lower case*. The keywords are known as *Reserved words*. Keywords or Reserved words are *Pre-defined identifiers*. 32 keywords are available in C.

Properties of Keywords

1. All the keywords in C programming language are defined in lowercase letters only.
2. Every keyword has a specific meaning; users can not change that meaning.
3. Keywords cannot be used as user defined names like variable, functions, arrays, pointers etc...
4. Every keyword in C programming language represents some kind of action to be performed by the compiler.

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

1.3.4. IDENTIFIERS

The names of variables, functions, labels and various other user-defined objects are called *Identifiers*. Identifiers are used for defining variable names, function names etc. The general rules to be followed when constructing an identifier are:

- 1) Identifiers are a sequence of characters. The only characters allowed are alphabetic characters, digits and the underscore character. Special characters are not allowed in identifier name.
- 2) The first character of an identifier is a letter or an underscore character.
- 3) Identifiers are case sensitive. For example, the identifiers TOTAL and Total are different.
- 4) Keywords are not allowed as identifier name.

VALID IDENTIFIERS:

Name area interest circum amount rate_of_interest sum

INVALID IDENTIFIERS

Identifiers	Reason
4 th	The first letter is a numeric digit.
int	The Keyword should not be a identifier
First name	Blank space is not allowed.

1.3.5. CONSTANTS

The data values are usually called as Constant. Constant is a quantity that does not change during program execution. This quantity can be stored at a location in the memory of the computer. *C has four types of constants: Integer, Floating, String and Character.*

INTEGER CONSTANT

An Integer Constant is an integer number. An integer constant is a sequence of digits. There are 3 types of integers namely decimal integer, octal integer and hexadecimal integer.

Decimal Integer consists of a set of digits 0 to 9 preceded by an optional + or - sign. Spaces, commas and non digit characters are not permitted between digits.

Example for valid decimal integer constants are : 123 -31 0 562321 + 78

Octal Integer constant consists of any combination of digits from 0 through 7 with a O at the beginning.

Examples of octal integers are : O26 O O347 O676

Hexadecimal integer constant is preceded by OX or Ox. They may contain alphabets from A to F or a to f. or a decimal digit (0 to 9). The alphabets A to F refers to 10 to 15 in decimal digits.

Example of valid hexadecimal integers are : OX2 OX8C OXbcd Ox123

REAL CONSTANT

Real constants are numbers with fractional part. Real constants are often called as *Floating Point constants*.

RULES

1. A real constant must have atleast one digit.
2. It must have a decimal point.
3. It could be either positive or negative.
4. Default sign is positive (If no sign)
5. Special characters are not allowed.
6. Omitting of digit before the decimal point, or digits after the decimal point is allowed. (Ex .655,12.)

Examples : +325.34 426.0 -32.76 -48.5792

Real Constants are represented in two forms:

- i) *Fractional form*
- ii) *Exponential form*

EXPONENTIAL FORM or SCIENTIFIC FORM

The Exponential form is used to represent very large and very small numbers. *The exponential form representation has two parts:* 1) *mantissa* and 2) *exponent*. The part appears before the letter ‘e’ is called *mantissa* and the part following the letter ‘e’ is called *exponent*, which represents a power of ten. The general form of exponential representation is.

mantissa e exponent

where “mantissa” is a decimal or integer quantity and the “exponent” is an integer quantity with an optional plus or minus sign.

The general rules regarding exponential forms are

1. The two parts should be separated by a letter “e” or “E”.
2. The mantissa and exponent part may have a positive or a negative sign.
3. Default sign of mantissa and exponent part is positive.
4. The exponent must have at least one digit.

Examples

The value 123.4 may be written as 1.234E+2 or 12.34E+1.

The value 0.01234 may be written as 1.234E-2 or 12.34E-3.

Differences between floating point numbers and integer numbers.

- Integer includes only whole numbers, but floating point numbers can be either whole or fractional.
- Integers are always exact, whereas floating point numbers sometimes can lead to loss of mathematical precision.
- Floating point operations are slower in execution and often occupy more memory than integer operations.

CHARACTER CONSTANTS

A character constant is either a single alphabet, a single digit or a single special character enclosed within a pair of single inverted commas.

The maximum length of a character constant can be 1 character

Examples: ‘A’ ‘1’ ‘\$’ ‘ ’ ‘;’

Each character constant has an integer value. This value is known as ASCII value. For example, the statement printf (“%d”, ‘A’) would print the value of 65. Because the ASCII value of A is 65.

Similarly the statement printf (“%c”, 65) would display the letter ‘A’.

STRING CONSTANTS

A combination of characters enclosed within a pair of double inverted commas is called as “String Constant”.

Examples: “Salem” “35.567” “\$125” “Rose”

Each string constant ends with a special character ‘\0’. This character is not visible when the string is displayed. Thus “salem” contains actually 6 characters. The ‘\0’ character acts as a string terminator. This character is used to find the end of a string.

Remember that a character constant (e.g., 'A') and the corresponding single-character string constant ("A") are not equivalent.

A character constant has an equivalent integer value, whereas a single-character string constant does not have an equivalent integer value and, in fact, consists of two characters - the specified character followed by the null character (\0).

1.3.6. VARIABLES – DEFINITION AND RULES

A quantity, which may vary during program execution, is called variable. Variables may be used to store a data value. Variables are actually memory locations, used to store constants. The variables are identified by names. The program may modify the values stored in a variable.

Rules for constructing variable names

1. A variable name is the combination of characters. The length of a variable depends upon the compiler.
2. The first character must be an alphabet or underscore.
3. Special characters like comma or blank are not allowed except an underscore character. The only characters allowed are letters, digits and underscore.
4. The variable name should not be a keyword.

Examples : area interest circumference fact date_of_birth

1.3.7. DECLARING VARIABLES

In C, all the variables used in a program are to be declared before they can be used. All variables must be declared in the beginning of the function. Type declaration statement is used to declare the type of various variables used in the program. The syntax for declaration is,

data_type variable_name(s)

where data-type is a valid data type plus any other modifiers. variable-name(s) may contain one or more variable names separated by commas.

Examples : int a, b, c; long int interest; unsigned char ch;

Declaration statement is used to allocate memory space for the variable. Declaration statement also provides a name for the location. Declaration statement declares that the program will use that variable name to identify the value stored at the location.

For example, the declaration

char sub-name

allocates a memory location of size one byte, of character type. This memory location is given a name of sub-name.

1.3.8. INITIALIZATION OF A VARIABLE (ASSIGNING VAUSES TO VARIABLES

The process of assigning initial values to variables is called initialization of variables. In C, an uninitialized variable can contain any garbage value. Therefore, the programmer must make sure all the variables are initialised before using them in any of the expressions. The value for a particular variable is initialized through declarative statement or assignment statement. For example to initialize the value 10 to an integer variable i, the following two methods are used.

- i) int i = 10;
- ii) int i;
i = 10;

The above two methods declares that i is an integer variable with an initial value of 10.

Examples : (i) float pi = 3.14; (ii) char alpha = "h"; (iii) int account = 10;

More than one variable can be initialized in a single statement. For example, the statement

```
int x = 1, y = 2, z;
```

declares x to be an int with value 1, y to be an int with value 2 and z to be an int of unpredictable value. Same value can be initialized to several variables with the single assignment statement.

Examples

```
int i, j, k, l, m, n;
float a, b, c, d, e, f;
i = j = k = l = m = n = 20;
a = b = c = d = e = f = 13.56;
```

1.3.9. DECLARING VARIABLES AS CONSTANTS

A constant is a quantity whose value does not change during program execution. The qualifier **const** is used to declare the variable as constant at the time of initialization. The general form to declare variables as constant is

```
const data type variable = value;
```

where const - keyword data type- valid type such as int, float etc.

Example : const float PI = 3.14;

const is a new data type qualifier. This tells the computer that the value of the **float** variable **PI** must not be modified by the program. But , it can be used on the right-hand side of an assignment like any other variable.

1.3.10. DECLARING VARIABLES AS VOLATILE

The qualifier **volatile** is used to tell explicitly the compiler that a variable's value may be changed at any time by some external sources not by the program(from outside the program). General form to declare a variable as volatile is

```
volatile data type variable;
```

where volatile -keyword data type - valid types such as int, float etc.

Example : volatile int x;

The value of x may be altered by some external factors even if it does not appear on the left-hand side of an assignment statement. When declaring a variable as volatile, the compiler will examine the value of the variable each time it is encountered to see whether any external alteration has changed the value.

1.3.11. DATA TYPES

C supports several data types. Each data type may be represented differently inside the computer's memory. There are four data types in C language. They are,

Types	Data Types
Basic data types	int, char, float, double
Enumeration data type	enum
Derived data type	pointer, array, structure, union
Void data type	void

Basic Data Types

Basic or Fundamental data types include the data types at the lowest level. i.e. those which are used for actual data representation in the memory. All other data types are based on the fundamental data types.

Examples: char, int, float, double.

int type is used to store positive or negative integers. float data type is used to store a single precision floating-point (real) number. Floating-point numbers are stored in 32 bits with 6 digits of precision. double data type is used to hold real numbers with higher storage range and accuracy than the type float. The data type char is used to store one character.

The size (number of bytes) and range of numbers to be stored in each data type is shown below.

Data Type	Size (Bytes)	Range
char	1	-128 to 127
int	2	-32,768 to 32,767
float	4	3.4 E-38 to 3.4 E+38
double	8	1.7 E-308 to 3.4 E+308

Derived Data Types

These are based on fundamental data types. i.e. a derived data type is represented in the memory as a fundamental data type.

Examples: pointers, structures, arrays.

Void data type :

1. void is an empty data type that has no value.
2. This can be used in functions and pointers.

1.3.12. DATA TYPES MODIFIERS(QUALIFIERS)

The basic data type may be modified by adding special keywords. These special keywords are called *data type modifiers (or) qualifiers*. Data type modifiers are used to produce new data types.

The modifiers are: *signed* *unsigned* *long* *short*

The above modifiers can be applied to integer and char types. Long can also be applied to double type.

Integer Type

A signed integer constant is in the range of -32768 to +32767. Integer constant is always stored in two bytes. In two bytes, it is impossible to store a number bigger than +32767 and smaller than -32768. Out of the two bytes used to store an integer, the leftmost bit is used to store the sign of the integer. So the remaining 15 bits are used to store a number. If the leftmost bit is 1, then the number is negative. If the leftmost bit is 0, then the number is positive.

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

Floating Point Type

Floating point number represents a real number with 6 digits precision. When the accuracy of the floating point number is insufficient, use the double to define the number. The double is same as float but with longer precision. To extend the precision further , use long double, which consumes 80 bits of memory space.

Character Type

Characters are usually stored in 8 bits of internal storage. The qualifier signed or unsigned can be explicitly applied to char. While unsigned characters have values between 0 and 255, signed characters have values from -128 to 127.

Size and Range of Data Types on 16 bit machine.

TYPE	SIZE (Bits)	Range
Char or Signed Char	8	-128 to 127
Unsigned Char	8	0 to 255
Int or Signed int	16	-32768 to 32767
Unsigned int	16	0 to 65535
Short int or Signed short int	8	-128 to 127
Unsigned short int	8	0 to 255
Long int or signed long int	32	-2147483648 to 2147483647
Unsigned long int	32	0 to 4294967295
Float	32	3.4 e-38 to 3.4 e+38
Double	64	1.7e-308 to 1.7e+308
Long Double	80	3.4 e-4932 to 3.4 e+4932

1.3.13. OVERFLOW AND UNDERFLOW OF DATA

Assigning a value which more than the upper limit of the data type is called overflow and less than its lower limit is called underflow.

In case of integer types, overflow results wrapping towards negative side and underflow results wrapping towards positive side.

In case of floating point types, overflow results +INF and underflow results -INF.

Example 1 :

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

```
{  
int a = 32770;  
printf( "%d",a);  
}
```

Output : - 32766

The range of integer is – -32768 to + 32767, assigning 32770 results overflow and wrap towards –negative side.

Example 2:

```
#include <stdio.h>  
void main()  
{  
int a = 33000;  
float b = 3.4e50;  
printf( "%d%f",a,b);  
}
```

Output : - 32536 +INF

The range of integer is – -32768 to + 32767, assigning 33000 results overflow and wrap towards –negative side and +INF is a result of float overflow.

1.3.14. COMMENTS

Comments are used to make a program more readable. Comments are not instructions. Comments are remarks written in a program. These remarks are used to give more information about the program. The compiler will ignore the comment statements.

In C, there are two types of comments.

1. **Single Line Comments:** Single line comment begins with // symbol. Any number of single line comments can be written.
2. **Multiple Lines Comments:** Multiple lines comment begins with /* symbol and ends with */. Any number of multiple lines comments can be included in a program.

In a C program, the comment lines are optional. All the comment lines in a C program just provide the guidelines to understand the program and its code.

Examples:

(e.g.) /* Program to find the Factorial */

Any number of comments can be placed at any place in a program. Comments cannot be nested. For example

/* Author Arul /* date 01/09/93 */ */ is not possible.

A comment can be split over more than one line. For example, the following is valid.

```
/* This statement is used to find the sum of two  
numbers */
```

1.3.15. ESCAPE SEQUENCES

Character combinations consisting of a backslash (\) followed by a letter or by a combination of digits are called "escape sequences." To represent a newline character, single quotation mark, or certain other characters in a character constant, escape sequences are used. An escape sequence is regarded as a single character and is therefore valid as a character constant.

- The escape sequence characters are also called as backslash character constants.
- These are used for formatting the output

For example, a line feed (LF), which is referred to as a *newline* in C, can be represented as \n. Such escape sequences always represent single characters, even though they are written in terms of two or more characters. The commonly used escape sequences are listed below.

ESCAPE CHARACTER	MEANING	ESCAPE CHARACTER	MEANING
bell (alert)	\a	carriage return	\r
backspace	\b	quotation mark ("")	\"
horizontal tab	\t	apostrophe ('')	\'
vertical tab	\v	question mark (?)	\?
newline (line feed)	\n	backslash	\\\
form feed	\f	null	\0

The following program outputs a new line and a tab and then prints the string *This is a test.*

```
#include <stdio.h>
int main()
{
    printf("\n\tThis is a test.");
    return 0;
}
```

Characteristics

- Although it consists of two characters, it represents single character.
- Every combination starts with back slash(\)
- They are non-printing characters.
- It can also be expressed in terms of octal digits or hexadecimal sequence.
- Each escape sequence has unique ASCII value.

1.3.16. SYMBOLIC CONSTANTS

- Names given to values that cannot be changed. Implemented with the #define preprocessor directive.

```
#define N 3000
#define FALSE 0
```
- Preprocessor statements begin with a # symbol, and are NOT terminated by a semicolon. Traditionally, preprocessor statements are listed at the beginning of the source file.

- Preprocessor statements are handled by the compiler (or preprocessor) before the program is actually compiled. All # statements are processed first, and the symbols (like N) which occur in the C program are replaced by their value (like 3000). Once this substitution has taken place by the preprocessor, the program is then compiled.
- In general, preprocessor constants are written in UPPERCASE. This acts as a form of internal documentation to enhance program readability and reuse.
- In the program itself, values cannot be assigned to symbolic constants.

Use of Symbolic Constants

- Consider the following program which defines a constant called TAXRATE.

```
#include <stdio.h>
#define TAXRATE 0.10
main () {
    float balance;
    float tax;
    balance = 72.10;
    tax = balance * TAXRATE;
    printf("The tax on %.2f is %.2f\n",balance, tax);}
```

1.4. C OPERATORS

1.4.1. INTRODUCTIONS TO OPERATORS

C has a very rich set of operators. So C language is sometimes called “the language of operators”. Operators are used to manipulate data and variables. An operator is a symbol, which represents some particular operation that can be performed on some data. The data itself (which can be either a variable or a constant) is called the ‘operand’.

Operators operate on constants or variables, which are called *operands*. Operators can be generally classified as either unary, binary or ternary operators. *Unary operators* act on one operand, *binary operators* act on two operands and *ternary operators* operate on three operands.

Depending on the function performed, the operators can be classified as

Arithmetic	Relational	Logical	Conditional	Assignment
Increment & Decrement	Modulo division	Bitwise	Special operators	

1.4.2. ARITHMETIC OPERATORS

Arithmetic operators are used to perform arithmetic operation in C. Arithmetic operators are divided into two classes: (i) Unary arithmetic operators and (ii) Binary arithmetic operators

Binary operators

Operator	Meaning
+	Addition or Unary Plus
--	Subtraction or Unary Minus
*	Multiplication
/	Division
%	Modulus Operator

Arithmetic Operators +, -, * and / can be applied to almost any built-in data types. Suppose that a and b are integer variables whose values are 10 and 3, respectively. Several arithmetic expressions involving these variables are shown below, together with their resulting values.

Operation	Value
a+ b	13
a-b	7
a* b	30
a/ b	3
a%b	1

Modulo operator: The modulo operator (%) gives the remainder of the division between the two integer values. For Modulo division, the sign of result is always that of the first operand or dividend.

For example, $13\%5 = 3$; $-13 \% - 5 = -3$; $-13\% 5 = -3$;
 $13\% 5 = 3$;

Example :

```
main()
{
int a, b,c,d;
a = 10;
b = 4;
c= a/b;
d = a % b;
printf("%d %d", c, d);
}
```

Modulo division operator cannot be used with floating point type. C does not have no option for exponentiation.

Unary minus operator

In unary minus operation, minus sign precedes a numerical constant, variable or an expression. A negative number is actually an expression, consisting of the unary minus operator, followed by a positive numeric constant. *Unary minus operation is entirely different from the subtraction operator (-). The subtraction operator requires two operands.*

1.4.3. INCREMENT AND DECREMENT OPERATORS

C contains two special operators ++ and --. ++ is called Increment operator. -- is called Decrement operator; The above two operators are called unary operators since they operate on only one operand. The operand has to be a variable and not a constant. Thus, the expression 'a++' is valid whereas '6++' is invalid.

Increment operators are used to increase the value of the variable by one and decrement operators are used to decrease the value of the variable by one in C programs.

- **Syntax:**

Increment operator: `++var_name;` (or) `var_name++;`

Decrement operator: `--var_name;` (or) `var_name--;`

- **Example:**

Increment operator : `++ i ; i ++ ;`

Decrement operator : `-- i ; i -- ;`

If the operator is used before the operand, then it is called *prefix operator*. (Example: `++a, --a`). If the operator is used after the operand, then it is called *postfix operator*. (Example: `a++, a--`).

Difference between pre/post increment & decrement operators in C:

Below table will explain the difference between pre/post increment and decrement operators .

Operator	Operator/Description
Pre increment operator (<code>++i</code>)	value of <code>i</code> is incremented before assigning it to the variable <code>i</code>
Post increment operator (<code>i++</code>)	value of <code>i</code> is incremented after assigning it to the variable <code>i</code>
Pre decrement operator (<code>-i</code>)	value of <code>i</code> is decremented before assigning it to the variable <code>i</code>
Post decrement operator (<code>i-</code>)	value of <code>i</code> is decremented after assigning it to variable <code>i</code>

Prefix and Postfix operators have the same effect if they are used in an isolated C statement. For example, the two statements

`x++;` and `++x;` have the same effect

But prefix and postfix operators have different effects when they are assigned to some other variable. For example the statements

`z = ++x;` and `z = x++;` have different effects.

Assume the value of `x` to be 10. The execution of the statement `z = ++x;` will first increment the value of `x` to 11 and assign new value to `z`. The above statement is equal to the following two statements.

```
x = x + 1 ;
z = x ;
```

The execution of the statement `z = x++;` will first assign the value of `z` to 10 and then increase the value of `x` to 11. The above statement is equal to

```
z = x ;
x = x + 1 ;
```

The decrement operators are also in a similar way, except the values of `x` and `z` which are decreased by 1.

Other Examples

`a = 10 , b=6`

`a=10, b = 6`

`c = a * b++`

`c = a * ++b`

Output:

`c = 60;`

`c = 70`

The expression `n++` requires a single machine instruction such as INR to carry out the increment operation. But `n+1` operation requires more instructions to carry out this operation. So the execution of `n++` is faster than `n+ 1` operation.

1.4.4. RELATIONAL OPERATORS

Relational operators are used to test the relationship between two operands. The operands can be variables, constants or expressions. C has six relational operators. They are

Operator	Meaning	Operator	Meaning
<	is less than	>=	is greater than or equal to
<=	is less than or equal to	==	is equal to
>	is greater than	!=	is not equal to

An expression containing a relational operator is called as a relational expression. The value of the relational expression is either true or false. If it is false, the value of the expression is 0 and if it is true, the value is 1.

Examples :

Suppose that i,j and k are integer variables whose values are 1, 2 and 3, respectively. Several relational expressions involving these variables are shown below.

Expression	Interpretation	Value
i < j	true	1
(1 + j) >= k	true	1
(j + k) > (i + 5)	false	0
k != 3	false	0
j == 2	true	1

1.4.5. LOGICAL OPERATORS

Logical operators are used to combine or negate expression containing relational expressions. C provides three logical operators.

Operator	Meaning
&&	Logical AND
	Logical OR
!	Logical NOT

Logical expression is the combination of two or more relational expressions. Logical operators are used to combine the result of evaluation of relational expressions. Like the simple relational expression, a logical expression also gives value of one or zero.

LOGICAL AND (&&)

This operator is used to evaluate two conditions or expressions simultaneously. If both the expressions to the left and to the right of the logical operator is true then the whole compound expression is true.

Example : a > b && x == 10

The expression to the left is a > b and that on the right is x == 10. The whole expression is true only if both expressions are true i.e., if a is greater than b and x is equal to 10.

LOGICAL OR (||)

The logical OR is used to combine two expressions or the condition. If any one of the expression is true, then the whole compound expression is true.

Example : $a < m \mid\mid a < n$

The expression evaluates to true if any one of them is true or if both of them are true. It evaluates to true if a is less than either m or n and when a is less than both m and n.

LOGICAL NOT (!)

The logical not operator takes single expression and evaluates to true if the expression is false and evaluates to false if the expression is true. In other words, it just reverses the value of the expression.

Examples

Suppose that i is an integer variable whose value is 7, f is a floating-point variable whose value is 5.5, and c is a character variable that represents the character 'w'. Several complex logical expressions that make use of these variables are shown below.

Expression	Interpretation	Value
$(i \geq 6) \&\& (c == 'w')$	true	1
$(i \geq 6) \mid\mid (c == 119)$	true	1
$(f < 11) \&\& (i > 100)$	false	0
$(c != 'p') \mid\mid ((i + f) \leq 10)$	true	1

The truth table for the logical operators is shown here using 1's and 0's.

p	q	$p \&\& q$	$p \mid\mid q$	$\neg p$
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

1.4.6. CONDITIONAL OPERATORS

Simple conditional operations can be carried out with the conditional operator (? :). *The conditional operator is used to build a sionditional expression.* The Conditional operator has two parts: ? and :

An expression that uses the conditional operator is called conditional expression. The conditional operator is a ternary operator because it operates on three operands. The general form is

expression1 ? expression 2 : expression 3 ;

The expression1 is evaluated first. If it is true (non – zero), then the expression 2 is evaluated and its value is returned. If expression 1 is false (zero), then expression 3 is evaluated and its value is returned. Only one expression either expression 2 or expression 3 will be evaluated at a time.

Conditional expression frequently appear on the righthand side of a simple assignment statement. The resulting value of the conditional expression is assigned to the identifier on the left.

Example : **max = c > d ? c : d;**

The purpose of the above statement is to assign the value of c or d to max, whichever is larger. First the condition ($c > d$) is tested. If it is true, $max = c$; if it is false $max = d$;

1.4.7. ASSIGNMENT AND SHORT-HAND ASSIGNMENT OPERATORS

The Assignment Operator (=) evaluates an expression on the right hand side of the expression and substitutes this value to the variable on the left hand side of the expression.

Example : **x = a + b ;**

Here the value of $a + b$ is evaluated and substituted to the variable x .

In addition, C has a set of shorthand assignment operators. Short hand assignment operators are used to simplify the coding of a certain type of assignment statement. The general form of this statement is

var oper = exp;

Here var is a variable, exp is an expression or constant or variable and $oper$ is a C binary arithmetic operator. The operator $oper =$ is known as shorthand assignment operator

The above general form translates to : $var = var \ op \ exp;$

The compound assignment statement is useful when the variable name is longer. For example

amount-of-interest = amount-of-interest * 10 ;

can be written as

amount-of-interest *= 10 ;

For example $a = a + 1;$ can be written as $a += 1$

The short hand works for all binary operators in C. Examples are

$x -= y;$	is equal to	$x = x - y;$
$x *= y;$	is equal to	$x = x * y;$
$x /= y;$	is equal to	$x = x / y;$
$x \% y;$	is equal to	$x = x \% y;$

1.4.8. BITWISE OPERATORS

The combination of 8 bits is called as one byte. A bit stores either a 0 or 1. Data are stored in the memory and in the registers as a sequence of bits with 0 or 1.

Bitwise operators are used for manipulation of data at bit level. *The operators that are used to perform bit manipulations are called bit operators.* C supports the following six bit operators.

Operator	Description	Operator	Description
&	Bitwise AND	~	One's Complement
	Bitwise OR	<<	Shift left
^	Bitwise X-OR	>>	Shift right

These operators can operate only on integers and not on float or double data types. All operators except \sim operator are binary operators, requiring two operands. When using the bit operators, each operand is treated as a binary number consisting of a series of individual 1s and 0s. The respective bits in each operand are then compared on a bit by bit basis and result is determined based on the selected operation.

Bitwise AND operator

Bitwise AND operator(&) is used to mask off certain bits. The result of ANDing operation is 1 if both the bits have a value of 1; otherwise it is 0. Assume the two variables a and b , whose values are 12 and 24. The result of the statement $c = a \& b$ is

a	0	0	0	0	1	1	0	0
b	0	0	0	1	1	0	0	0
c=a&b	0	0	0	0	1	0	0	0

The value of c is 8. To mask the particular bit (s) in a value, AND the above value with another value by placing 0 in the corresponding bit in the second value.

Bitwise OR operator Bitwise OR operator(`||`) is used to turn ON certain bits. The result of ORing operation is 1 if any one of the bits have a value of 1; otherwise it is 0. Assume the two variables `a` and `b`, whose values are 12 and 24. The result of the statement `c = a | b` is

a	0	0	0	0	1	1	0	0
b	0	0	0	1	1	0	0	0
c=a b	0	0	0	1	1	1	0	0

The value of c is 28. To turn ON a particular bit(s) in a pattern of bits, OR the above value with another value by placing 1 in the corresponding bit in the second value.

Bitwise Exclusive OR operator

The result of bitwise Exclusive OR operator (^) is 1 if only one of the bits is 1; otherwise it is 0. The result of $a \wedge b$ is

A	0	0	0	0	1	1	0	0
b	0	0	0	1	1	0	0	0
c = a^b	0	0	0	1	0	1	0	0

The value of c is 20.

Bitwise Complement operator

The complement operator \sim complements all the bits in an operand. That is, 0 changes to 1 and 1 changes to 0. If the value of a is 00001100, then $\sim a$ is 11110011.

Shift operators

The shift operators are used to move bit pattern either to the left or right. The general forms of shift operators are

Left shift : operand $\ll n$ Right shift : operand $\gg n$

where *operand* is an integer and *n* is the number of bit positions to be shifted. The value for *n* must be an integer quantity.

Assume the value of a is 12. Then the result of `a << 2` will be follows:

a	0	0	0	0	1	1	0	0
a<<2	0	0	1	1	0	0	0	0

The value of c is 48. The above operation shifts the bits to the left by two places and the vacated places on the right side will be filled with zeros. **Every shift to the left by one position corresponds to multiplication by 2.** Shifting two positions is equal to multiplication by 2×2 i.e., by 4. Similarly, every shift to the right by one position corresponds to division by 2.

1.4.9. SPECIAL OPERATORS

C supports some special operators like comma operator, sizeof operator, pointer operators (& and *) and member selection operators (. And ->).

Comma Operator:

Comma operator is used in the assignment statement to assign many values to many variables.

Example: int a=10, b= 20, c= 30;

Comma operator is also used in for loop in all the three fields.

Example: for (i=0, j=2; i<10; i=i+1;j=j+2);

Sizeof Operator:

Another unary operator is the sizeof operator . This operator returns the size of its operand, in bytes. This operator always precedes its operand. The operand may be a variable, a constant or a data type qualifier. Consider the following program.

```
main(){
    int sum;
    printf("%d", sizeof(float));
    printf("%d", sizeof (sum));
    printf("%d", sizeof (234L));
    printf("%d", sizeof ('A'));
}
```

Here the first printf() would print out 4 since a float is always 4 bytes long. With this reasoning, the next three printf() statements would output 2, 4 and 2. Consider an array school[]="National". Then, sizeof (school) statement will give the result as 8.

1.4.10. HIERARCHY OF OPERATIONS (PRECEDENCE AND ACCOCIATIVITY)

The order in which the operations are performed in an expression is called hierarchy of operations. The priority or precedence of operators is given below.

The arithmetic operators have the highest priority. Both relational and logical operators are lower in precedence than the arithmetic operators (except !). The shorthand assignment operators have the lowest priority.

Order	Category	Operator	Operation	Associativity
1	Highest precedence	() [] → :: ·	Function call	L → R Left to Right
2	Unary	! ~ + - ++ -- & * Size of	Logical negation (NOT) Bitwise 1's complement Unary plus Unary minus Pre or post increment Pre or post decrement Address Indirection Size of operand in bytes	R → L Right -> Left
3	Multiplication	*	Multiply	L → R
		/	Divide	
		%	Modulus	
4	Additive	+	Binary Plus	L → R
		-	Binary Minus	

5	Shift	<< >>	Left shift Right shift	$L \rightarrow R$
6	Relational	< <= > >=	Less than Less than or equal to Greater than Greater than or equal to	$L \rightarrow R$
7	Equality	== !=	Equal to Not Equal to	$L \rightarrow R$
8	Bitwise AND	&	Bitwise AND	$L \rightarrow R$
9	Bitwise XOR	^	Bitwise XOR	$L \rightarrow R$
10	Bitwise OR		Bitwise OR	$L \rightarrow R$
	Logical AND	&&	Logical AND	$L \rightarrow R$
12	Conditional	? :	Ternary Operator	$R \rightarrow L$
13	Assignment	= *= %= /= += -= &= ^= = <<= >>=	Assignment Assign product Assign remainder Assign quotient Assign sum Assign difference Assign bitwise AND Assign bitwise XOR Assign bitwise OR Assign left shift Assign right shift	$R \rightarrow L$
14	Comma	,	Evaluate	$L \rightarrow R$

An expression within the parentheses is always evaluated first. *One set of parentheses can be enclosed within another. This is called nesting of parentheses.* In such cases, innermost parentheses is evaluated first.

Associativity means how an operator associates with its operands. For example, the unary minus associated with the quantity to its right, and in division the left operand is divided by right. The assignment operator '=' associates from right to left. Associativity also refers to the order in which C evaluates operators in an expression having same precedence. For example, the statement $a = b = 20 / 2$; assigns the value of 10 to b, which is then assigned to 'a', since associativity said to be from right to left.

1.4.11. EVALUATION OF EXPRESSIONS

Example 1: Determine the hierarchy of operations and evaluate the following expression:

$$i = 2 * 3 / 4 + 4 / 4 + 8 - 2 + 5 / 8$$

Stepwise evaluation of this expression is shown below:

```
i = 2 * 3 / 4 + 4 / 4 + 8 - 2 + 5 / 8
i = 6 / 4 + 4 / 4 + 8 - 2 + 5 / 8 operation: *
i = 1 + 4 / 4 + 8 - 2 + 5 / 8 operation: /
i = 1 + 1 + 8 - 2 + 5 / 8 operation: /
i = 1 + 1 + 8 - 2 + 0 operation: /
i = 2 + 8 - 2 + 0 operation: +
```

i = 10 - 2 + 0 operation: +

i = 8 + 0 operation : -

i = 8 operation: +

Note that $6 / 4$ gives 1 and not 1.5. This so happens because 6 and 4 both are integers and therefore would evaluate to only an integer constant. Similarly $5 / 8$ evaluates to zero, since 5 and 8 are integer constants and hence must return an integer value.

Example 2: Determine the hierarchy of operations and evaluate the following expression:

k = 3 / 2 * 4 + 3 / 8 + 3

k = 3 / 2 * 4 + 3 / 8 + 3

k = 1 * 4 + 3 / 8 + 3 operation: /

k = 4 + 3 / 8 + 3 operation: *

k = 4 + 0 + 3 operation: /

k = 4 + 3 operation: +

k = 7 operation +

1.4.12. ASSIGNMENT STATEMENTS

Assignment statements are used to assign the values to variables. Assignment statements are constructed using the assignment operator (=). General form of assigning value to variable is

variable = expression;

where expression is a constant or a variable name or the expression (Combinations of constants, variables and operators).

Examples : (i) a = 5; (ii) a = b; (iii) a = a + b; (iv) a = a > b;

Rules to be followed when constructing assignment statements

1. Only one variable is allowed on the left hand side of '=' expression $a = b * c$ is valid, whereas $a + b = 5$ is invalid.
2. Arithmetic operators can be performed on char, int, float and double data types. For example the following program is valid, since the addition is performed on the ASCII value of the characters x and y.

```
char a,b;  
int z;  
a = 'x';  
b = 'y'  
z = a +b;
```

3. All the operators must be written explicitly. For example $D = x. y .z$ is invalid. It must be written as $D = x * y * z$.

Multiple Assignment Statement

The same value can be assigned to more than one variable in a single assignment. This is possible by using multiple assignments. For example to assign the value 30 to the variable a, b, c, d is

$a = b = c = d = 30;$

However, this cannot be done at the time of declaration of variables.

For example

$\text{int } a = b = c = d = 30$ is invalid.

1.4.1. EXPRESSIONS

An expression is the combination of Variables and Constants connected by any one of the arithmetic operator. Examples for expressions are:

$$(i) \quad a + b \quad (ii) \quad b+5 \quad (iii) \quad a+b*5-6/c \quad (iv) \quad 8.2+ a/b+6.7$$

Integer expression: If all the variables and constants in an expression are of integer type, then this type of expression is called as integer expression. An integer expression will always give a result in integer value.

Real Expression: If all variables and constants in an expression are of real type, then this type of expression is called as real expression, A real expression will always give a result in real value.

Mixed mode Expression: If the elements in an expression are of both real and integer types, then this type of expression is called as mixed mode expression. A mixed mode expression will always give a result in real value.

Examples:

Integer Expression	Real Expression	Mixed Mode Expression
int a,b,c; c = a+b/5 + c/a;	float a,b,c; c = a+b/2.0 + c+ 5.6	int a,b; float c,d; d = 5 + a/c + c/b + 6.1

1.4.14. TYPE CONVERSION

Implicit type conversion

C permits mixing of constants and variables of different types in an expression. C automatically converts any intermediate values to the proper type so that the expression can be evaluated without losing any significance. This automatic type conversion is known as *implicit type conversion*

When constants and variables of different types are mixed in an expression, they are all converted to the same type. The compiler converts all operands up to the type of the largest operand, which is called *type promotion*. The order of various data type is

char < int < long < float < double

For example if an expression contains an operator between int and float, the int would be automatically promoted to a float before carrying out the operation. The following table shows the final data type when two operands are of different types.

Opearnd1	Opearnd2	Result
char	int	Int
char	float	float
int	float	float
int	char	Int
long int	float	fFloat
float	double	double
double	char	double
double	long int	double

Explicit Conversion (Type cast operator)

Consider for example the ratio of number of female and male students in a class is

$$\text{Ratio} = \text{female_students} / \text{male_students}$$

Since if `female_students` and `male_students` are declared as integers, the decimal part will be rounded off and its ratio will represent a wrong figure. This problem can be solved by converting locally one of the variables to the floating point as shown below.

$$\text{Ratio} = (\text{float}) \text{female_students} / \text{male_students}$$

The operator `float` converts the `female_students` to floating point for the purpose of evaluation of the expression. Then using the rule of automatic conversion, the division is performed by floating point mode, thus retaining the fractional part of the result.

The process of such a local conversion is known as explicit conversion or casting a value. The general form is

(type_name) expression

1.5. I/O FUNCTIONS

C has no input and output statement to perform the input and output operations. C language has a collection of library functions for input and output (I/O) operations. The input and output functions are used to transfer the information between the computer and the standard input / output devices.

Console I/O functions - functions to receive input from keyboard and write output to VDU.

1.5.1. FORMATTED AND UNFORMATTED FUNCTIONS

The basic difference between formatted and unformatted I/O functions is that the formatted functions allow the input read from the keyboard or the output displayed on the VDU to be formatted as per our requirements. For example, if values of average marks and percentage marks are to be displayed on the screen, then the details like where this output would appear on the screen, how many spaces would be present between the two values, the number of places after the decimal points etc., can be controlled using formatted functions.

Console Input/Output functions						
Formatted functions			Unformatted functions			
Type	Input	Output	Type	Input	Output	
char	scanf()	printf()	char	getch()	putch()	
				getchar()	putchar()	
				getche()		
int	scanf()	printf()	int	-	-	
float	scanf()	printf()	float	-	-	
string	scanf()	printf()	string	gets()	puts()	

1.5.2. PRINTF () FUNCTION

`printf ()` function is used to display data on the monitor. The `printf ()` function moves data from the computer's memory to the standard output device. The general format of `printf ()` function is

```
printf ( "control string", list of arguments );
```

Comma is used to separate the control string from variable list.

The control string can contain :

- the characters that are to be displayed on the screen.
- format specifiers that begin with a % sign.
- escape sequences that begin with a backward slash (\) sign.

Examples

```
printf ("%f, %f, %d, %d", i, j, k + i, 5);  
printf ("Sum of %d and %d is %d", a, b, a + b );  
printf ("Two numbers are %d and %d", a, b );
```

printf () never supplies a newline automatically. Therefore multiple printf () statements may be used to display one line of output. A new line can be introduced by the new line character \n.

Conversion Characters or Format Specifiers

The character specified after % is called a conversion character. Conversion character is used to convert one data type to another type.

Conversion Character	Meaning
%c	Data item is displayed as a single character
%d	Data item is displayed as a signed decimal integer
%e	Data item is displayed as a floating-point value with an exponent
%f	Data item is displayed as a floating-point value without an exponent
%i	Data item is displayed as a signed decimal integer
%o	Data item is displayed as an octal integer, without a leading zero
%s	Data item is displayed as a string
%u	Data item is displayed as an unsigned decimal integer
%x	Data item is displayed as a hexadecimal integer, without the leading Ox

List of Variable Arguments

The arguments may be constants, single variable or array name or more complex expressions.

1.5.3. FORMATTED PRINTF FUNCTIONS

Formatted printf function is used for the following purposes:

1. To print values at the particular position of the screen.
2. To insert the spaces between the two values.
3. To give the number of places after the decimal point.

The above things can be achieved by adding modifier to the format specifiers. Modifiers are placed between the percent sign (%) and the format specifier. A maximum field width, the number of decimal places, left justification can be specified by using the modifiers.

The number placed between percent sign (%) and the format specifier is called a field width specifier.

For outputting Integer Numbers

The general format of the field width specifier is

```
%Wd
```

where W is the minimum field width. If the value to be printed is greater than the specified field width, the whole value is displayed. If the output is shorter than the length specified, remaining spaces

will be filled by blank spaces and the value is right justified. The following example illustrates the output of the number 1234 under different formats.

Format	Output					
printf("%d",1234)	1	2	3	4		
printf("%6d",1234)			1	2	3	4
printf("%2d",1234)	1	2	3	4		
printf("%-6d",1234)	1	2	3	4		
printf("%4d",-1234)	-	1	2	3	4	
printf("%06d",1234)	0	0	1	2	3	4

Commonly used Flags:

<input type="checkbox"/> -	Data item is left justified within the field
<input type="checkbox"/> +	A sign (either + or -) will precede each signed numerical data item. Without this flag, only negative data items are preceded by a sign.
<input type="checkbox"/> 0	Causes leading zeros to appear instead of leading blanks.

Outputting of Real Numbers

The general format for field width specification for outputting real number is

% W.d f

Where W represents the minimum number of width used for displaying the output value and "d" indicates the number of digits to be displayed after the decimal point. *This number is called precision.* The precision is an unsigned number. The given value is rounded to d decimal places and printed right-justified in the field of W columns. The default precision is 6 decimal places. Real numbers are also displayed in exponentiation form by using the specification e.

The following example illustrates the output of the number y = 12.3456 under different formats.

printf("%8.4f",y)	1	2	.	3	4	5	6
-------------------	---	---	---	---	---	---	---

printf("%8.2f",y)			1	2	.	3	5
-------------------	--	--	---	---	---	---	---

printf("%-8.2f",y)	1	2	.	3	5		
--------------------	---	---	---	---	---	--	--

printf("%f",y)	1	2	.	3	4	5	6	0	0
----------------	---	---	---	---	---	---	---	---	---

printf("%8.2e",y)	1	.	2	3	e	+	0	1	
-------------------	---	---	---	---	---	---	---	---	--

Outputting of string data

The general format for field width specification for outputting string data is

% W. d s

where W represents the minimum number of width used for displaying the output value and d indicates the maximum number of characters that can be displayed. If the precision specification is less than the total number of characters in the string, the excess rightmost characters will not be displayed.

When using string, for the minimum field width specification, leading blanks will be added if the string is shorter than the specified field width. Additional space will be allocated if the string is larger than the specified field width.

Example

```
main(){  
char item[9] = ""computer"";  
printf(%5s\n%11s\n%11.5s\n%.5s",item,item,item,item);}
```

Output:

c	o	m	P	U	t	e	r
C	O	m	p	u	t	e	r
		c	o	m	p	u	
c	o	m	p	U			

1.5.4. SCNF() FUNCTION

scnf() function is used to read the value from the input device. The general format of scnf () function is

```
scanf ("Control string", list of address of arguments );
```

The list of address of arguments represents the data items to be read. Each variable name must be preceded by an ampersand (&). The arguments are actually pointer which indicate where the data items are stored in the memory. Example of a scnf function is

```
scanf ( "%d %f %c" , &a, &b, &ch);
```

The control string consists of format specifier, white space character and non-white space character. The format specifier in scnf are very similar to those used with printf ().

Important points while using scnf are

1. Every basic data type variable must be preceded by (&) sign. **In the case of string and array data type, the data name is not preceded by the character &.**
2. Text to be printed is not allowed in scnf statement. For example *scanf ("Enter the number %d", &a);* is not valid.
3. The data items must correspond to the arguments.
4. If two or more data items are entered, they must separated by white space characters. (blank spaces, tabs or new line characters). Data items may continue into two or more lines. For example, for the statement

```
scanf ("%s %d %f", name, &regno, &avg );
```

the data items can be entered in the following methods.

- | | | | |
|----------|-----------------|-------------|-----------------------|
| (i) Arul | (ii) Arul 12345 | (iii) Arul | (iv) Arul 12345 85.56 |
| 12345 | 85.56 | 12345 85.56 | |
| 85.56 | | | |

Modifier with scnf functions

A modifier is used to specify the maximum field length. The modifier is placed between % sign and the format specifier.

Example : scanf ("%6s", str1);

is used to read a string of maximum 6 characters. If the input is more than 6 characters, first 6 characters will be assigned.

Length of data items may be lesser than the specified field width. But the number of characters in the actual data item cannot exceed the specified field width. Any excess character will not be read. Such uncovered characters may be incorrectly interpreted. This excess character acts as the input for the next data items.

1.5.5. UNFORMATTED FUNCTIONS

GETCHAR () FUNCTION

getchar() function is used to read one character at a time from the standard input device. When the getchar() function is called, it waits until a key is pressed and assigns this character as a value to getchar function. The value is also echoed on the screen.

The getchar() function does not require any argument. But a pair of empty parentheses must follow the word getchar. In general, the getchar function is written as

```
variable_name = getchar();
```

where variable_name is a valid C identifier that has been declared as char type.

For example,

```
char letter;  
letter = getchar();
```

will assign the character ‘A’ to the variable letter when pressing A on the keyboard. getchar () accepts all the characters upto the pressing of enter key, but reads the first character only.

PUTCHAR() FUNCTION

Single character can be displayed using the function putchar (). The function putchar () stands for “put character” and uses a argument. The general form of the putchar() function is

```
putchar (argument);
```

The argument may be a character variable or an integer value or the character itself contained within a single quote.

Examples

```
void main() {  
    char x = "A";  
    putchar (x);  
    putchar ("B");  
}
```

gets() function

gets() accepts any line of string including spaces from the standard Input device (keyboard). gets() stops reading character from keyboard only when the enter key is pressed.

Syntax for gets()

```
gets(variable_name);
```

puts() function

puts displays a single / paragraph of text to the standard output device.

Syntax for puts in C :

```
puts(variable_name);
```

Sample program :

```
#include<stdio.h>
#include<conio.h>
void main()
{
char a[20];
gets(a);
puts(a);
getch();
}
```

1. **gets** is used to receive user input to the array. **gets** stops receiving user input only when the **Newline character (Enter Key)** is interrupted.
2. **puts** is used to display them back in the monitor.

ADDITIONAL PROGRAMS

1. Assume all variables are of type int. Find the value of each of the following variables:

- a) $x = (\text{int}) 3.8 + 3.3;$ b) $x = (2 + 3) * 10.5;$ c) $x = 3 / 5 * 22.0;$ d) $x = 22.0 * 3 / 5;$
- a) 6 (reduces to $3 + 3.3$) b) 52 c) 0 (reduces to $0 * 22.0$) d) 13 (reduces to $66.0 / 5$)

2. Construct statements that do the following:

- a. Decrease the variable x by 1.
- b. Assigns to m the remainder of n divided by k .
- c. Divide q by b minus a and assign the result to p .
- d. Assign to x the result of dividing the sum of a and b by the product of c and d .

- a) $x--;$ or $--x;$ or $x = x - 1;$ b) $m = n \% k;$ c) $p = q / (b - a);$ d) $x = (a + b) / (c * d);$

3. Construct an expression to express the following conditions:

- a. number is equal to or greater than 90 but smaller than 100.
- b. ch is not a q or a k character.
- c. number is between 1 and 9 (including the end values) but is not a 5.
- d. number is not between 1 and 9.

- a. $\text{number} \geq 90 \ \&\& \ \text{number} < 100$ b. $\text{ch} \neq 'q' \ \&\& \ \text{ch} \neq 'k'$
c. $(\text{number} \geq 1 \ \&\& \ \text{number} \leq 9) \ \&\& \ \text{number} \neq 5$ d. $\text{number} < 1 \ || \ \text{number} > 9$

- 4. Given that 39.37 inches are equivalent to 1 meter, write a program that converts a given number of inches to the equivalent length of centimeters.**

```
#include<stdio.h>
void main()
{
float inches,cm;
clrscr();
printf("Enter the value in inches ");
scanf("%f",&inches);
cm = (inches/39.37)*100;
printf("\n The %.2f inches is equivalent to %.2f centimeter", inches,cm);
getch();
}
```

- 5. Write a C program to print out the square and cube of any integer value from the terminal.**

```
#include<stdio.h>
#include<string.h>
void main()
{
int number,square,cube;
printf(" Enter the value :");
scanf("%d",&number);
square = number*number;
cube = number*number*number;
printf("\n\n The square of given number is:%d",square);
printf("\n\n The cube of given number is:%d",cube); }
```

- 6. Write a C program to convert a given character from uppercase to lowercase and vice versa by using bitwise operator.**

```
int main()
{
char input;
printf("Character to convert: ");
scanf("%c",&input); //read character
printf("Converted character: %c", input ^ 32);
}
```

ASCII values of the uppercase and lowercase characters have a difference of 32. For example, in ASCII, "A" is represented by 65_{10} while "a" is represented by 97_{10} ($97-65 = 32$). At the bit level, only difference between the two characters is the 5th bit.

```
65 = 0 1 0 0 0 0 1
97 = 0 1 1 0 0 0 1
32 = 0 0 1 0 0 0 0
```

Therefore by inverting the 6th bit of a character it can be changed from uppercase to lowercase and vice versa. Bitwise XOR operation can be used to invert bits. Therefore any ASCII value XOR with 32 will invert its case from upper to lower and lower to upper.

- 7. Temperatures of a city in Fahrenheit degree are input through a keyboard. Write a program to convert the temperature into centigrade degrees**

```
#include <stdio.h>
#include <conio.h>
void main()
{
    float fahr,cel;
    clrscr();
    printf("\n Let us convert farenheit to celcius:");
    printf("\n Enter the temperature in farenheit\n");
    scanf("%f",&fahr);
    cel= 5*(fahr-32)/9.0;
    printf("\nThe value in celcius is : %f", cel);
    getch();
}
```

- 8. Using ternary (conditional) operator, write a C program to find the absolute value of a number.**

```
#include <stdio.h>
#include <conio.h>
void main(){
    int num,r;
    clrscr();
    printf(" Enter a number");
    scanf("%d",&num);
    r=(num>0)?(num):(-num);
    printf("The absolute value of %d is %d",num,r);
    getch(); }
```

- 9. `printf(""%-10s"", ""ABDUL"""); displays (A) ABDULbbbb (B) bbbbABDUL (C) ABDULbbbbbb (D) bbbbbbbbbbABDUL`**

Ans: A : -10s will print ABDUL in 10 space ABDUL followed by 5 blank space.

- 10. Read two float numbers and find sum, difference, product and division of the above numbers.**

```
void main() { float x,y,sum,difference,product,division;
printf("ENTER TWO NUMBERS=\n");
scanf("%f%f",&x,&y);
sum=x+y;
difference=x-y;
product=x*y;
division=x/y;
printf("\n\tSum=%f\tDifference=%f\n\tProduct=%f\tDivision=%f",sum,difference,product,division);
}
```

- 11. Write a program to read the price of an item in decimal form (like 15.95) and print the output in paisa (like 1595 paisa).**

```
void main() {
int b;
```

```

float a;
a=15.95;
clrscr();
b=100*a;
printf("%d",b); }

```

- 12.** Given the values of the variables X,Y and Z write a program to rotate their values such that X has the value of Y,Y has the value of Z and Z has the value of X.

```

void main(){
int x,y,z,temp;
printf("Enter the value of x,y,z\n");
scanf("%d %d %d",&x,&y,&z);
temp=x;
x=y;
y=z;
z=temp;
printf("%d %d %d",x,y,z);}

```

- 13.** Write a program that reads a floating-point number and then displays right-most digit of the integral part of the number.

```

void main() { int a,e;
float p;
printf("Enter the value of p\n");
scanf("%f",&p);
a=int(p);
printf("%d\n",a);
e=a%10;
if(a>10)
printf("%d\n",e); }

```

- 14.** Write a program to print the value 345.6789 in fixed-point format with the following specifications: (a) correct to two decimal place, (b)correct to five decimal place and (c)correct to zero decimal place.

```

#include
#include
void main(){
float a=345.6789;
printf("The two decimal place is: %.2f\n",a);
printf("The five decimal place is: %.5f\n",a);
printf("The zero decimal place is: %.0f",a);
getch(); }

```

- 15.** What conversion specification would you use to print each of the following?

- A decimal integer with a field width equal to the number of digits
- A hexadecimal integer in the form 8A in a field width of 4
- A floating-point number in the form 232.346 with a field width of 10
- A floating-point number in the form 2.33e+002 with a field width of 12
- A string left-justified in a field of width 30

ANSWER : a) %d b) %4X c) %10.3f d) %12.2e e) %-30s

SUMMARY

- program in any programming language. Generally, program development life cycle contains the following 6 phases Problem Definition , Problem Design , Coding , ,Testing , Documentation , Maintenance.
- A computer program is a sequence of instructions written to perform a specified task with a computer.
- Programming language is a set of grammatical rules for instructing a computer to perform specific tasks.
- Low Level languages are divided in to Machine language and Assembly language.
- Features of the High Level Languages are(i) Simplicity (ii) Naturalness: (iii) Abstraction: (iv) Efficiency: (v) Efficiency: (vi) Compactness (vii) Locality (viii) Extensibility.
- Algorithm is a step-by-step method of solving a problem or making decisions.
- A flowchart is a diagrammatic representation that illustrates the sequence of operations to be performed to get the solution of a problem.
- C has all the advantages of assembly language and all the significant features of modern high-level language. So it is called a “Middle Level Language”.
- The C programming language is a structure oriented programming language, developed at Bell Laboratories in 1972 by Dennis Ritchie
- The Features of C programming language are Reliability, Portability, Flexibility, Interactivity, Modularity, Efficiency and Effectiveness.
- All C programs are having the following sections/parts : Documentation section , Link Section, Definition Section, Global declaration section, Function prototype declaration section, Main function, User defined function definition section
- A statement causes the compiler to carry out some action. There are 3 different types of statements – expression statements compound statements and control statements. Every statement ends with a semicolon.
- Statement may be single or compound (a set of statements). Most of the statements in a C program are expression statements.
- C language character set contains the following set of characters... Alphabets ,Digits , Special Symbols
- Every C program is a collection of instructions and every instruction is a collection of some individual units. Every smallest individual unit of a c program is called token.
- Keywords are pre-defined words in a C compiler. Each keyword is meant to perform a specific function in a C program. C language supports 32 keywords.
- The value of the constants can not be modified by the program once they are defined. Constants refer to fixed values. The different types of constants are : (i) Integer (ii) Real (iii) Character (iv) String
- A constant is an entity that doesn't change whereas a variable is an entity that may change.
- Identifier is a collection of characters which acts as name of variable, function, array, pointer, structure, lable etc...
- A single character enclosed within a pair of single quotes is called single character constant. Example : 'A' . Sequence of characters enclosed in double quotes. Example:"SALEM"
- Data used in c program is classified into different types based on its properties. In c programming language, data type can be defined as a set of values with similar characteristics.

All the values in a data type have the same properties. The memory size and type of value of a variable are determined by variable data type.

- Primary data types are also called as Built-In datatypes. The following are the primary datatypes in c programming language Integer Data type , Floating Point Datatype ,Double Datatype, Character Data type
- The void data type means nothing or no value. Generally, void is used to specify a function which does not return any value
- Derived data types are user-defined data types. The derived datatypes are also called as user defined data types or secondary data types. In c programming language, the derived datatypes are created using the following concepts...Arrays , Structures ,Unions and Enumeration
- An operator is a symbol, which represents some particular operation that can be performed on some data. Operators operate on constants or variables, which are called operands.
- These C operators join individual constants and variables to form expressions.
- A Modulus operator gives the remainder value. This operator is applied only to integral operands and cannot be applied to float or double.
- Operators, functions, constants and variables are combined together to form expressions.
- Consider the expression $A + B * 5$. where, +, * are operators, A, B are variables, 5 is constant and $A + B * 5$ is an expression.
- If all the variables and constants in an expression are of integer type, then this type of expression is called as integer expression.
- Bit operators are used to perform bit operations. Bit wise operators in C language are & (bitwise AND), | (bitwise OR), ~ (bitwise OR), ^ (XOR), << (left shift) and >> (right shift).
- These operators are used to perform logical operations on the given expressions. There are 3 logical operators in C language. They are, logical AND (&&), logical OR (||) and logical NOT (!).
- Conditional operators return one value if condition is true and returns another value if condition is false. This operator is also called as ternary operator.
- Operators that act upon a single operand to produce a new value are called unary operator.
- C supports special operators like comma operator, size of operator, pointer operators (& and *) and member selection operators (. and ->).
- The priority or precedence in which the operations in an arithmetic statement are performed is called the hierarchy of operations.
- When there are more than one operator with same precedence [priority] then we consider associatively , which indicated the order in " which the expression has to be evaluated. It may be either from Left to Right or Right to Left.
- The combination of backslash(\) and some character which together represent one character are called escape sequences. Ex.: '\n' represents newline character.
- The comment lines are simply ignored by the compiler. They are not executed. In C, there are two types of comments. (i) Single line comments (ii) Multi line comments

5.

REVIEW QUESTIONS AND PROGRAMS

PART – A (2 Marks)

1. Define the term Program.
2. What are the different phases available in program development life cycle.
3. What are the different types of errors ?
4. Define Programming Language.
5. Give the different types of programming languages.
6. Write down the advantages and disadvantages of machine language.
7. Write down the advantages and disadvantages of assembly language.
8. Define Algorithm and Flow Chart.
9. List down the steps in executing a C Program.
10. Why C Language is called as Middle Level Language.
11. Define the term constant. List down the various types of constants.
12. What are the rules to be followed when constructing a floating point constant?
13. What are the data qualifiers? List them.
14. What is the output of the following program?

```
main() {  
    int x = 10;  
    printf ("%d %d", ++x, ++x);}
```
15. What is the numerical value of the expression $x \geq y \mid\mid y > x$?
16. Without using a conditional statement how to find whether a given number is odd or even?
17. Using a conditional statement , write a statement to find the biggest number between given three numbers a , b and c.
18. What is the difference between a statement and a block?
19. Why do we need different data types?
20. Define the term “ Token”. Give some examples.
21. $s++$ or $s = s+1$, which can be recommended to increment the value by 1 and why?
22. Define the term operator? How the operators are classified?
23. What is initialization? Why it is important?
24. What is a variable and what is the “value” of the variable?
25. When dealing with very small and very large numbers, what steps would you take to improve the accuracy of the calculations?

PART – B (3 Marks)

1. What are the features of C Language? Explain
2. What are the characteristics of an algorithm? Explain
3. What are symbols used in flow chart? Explain the purpose of each symbol.

4. What are the limitations of using flow chart? Define them.
5. Write down the history of C Language.
6. Mention any six features of C Language.
7. What is a keyword? What are the features of Keyword? How many keywords are available in C Language?
8. What is typecasting? Give an example.
9. Write a c program to add two numbers without using addition operator
10. Name and describe the basic data types.
11. State the use of const and volatile qualifiers.
12. What do you mean by initialization of a variable? Give examples.
13. What are unary operators? State the purpose of each.
14. What do you mean by hierarchy of operations? Give an example.
15. Why do we use comments in programs? What are the two types of comments?
16. What do the getchar() and putchar() functions do ?

PART – C (5 Marks/ 10 Marks)

1. What is program development life cycle? With a neat diagram, explain different phases of program development life cycle.
2. Explain the general structure of C Program.
3. With a grammatical representation, explain the various steps involved in executing a C Program.
4. Briefly explain about the various types of constants used in C Language with examples.
5. Explain about the different types of data types available in C language?
6. Explain type casting with an example program
7. Briefly explain about the precedence and associativity of operators.
8. State the use of conditional operator with an example.
9. What are the special operators available in C language? Explain them
10. Explain different types of operators available in C language? List them with their uses.
11. State the difference between post increment and pre increment operators? Explain your answer by giving examples.
12. Describe the characteristics and purpose of escape sequence characters.
13. Write a program that will obtain the length and width of a rectangle from the user and calculate its area and perimeter.
14. Write a program to print the size of various data types in C.
15. Given the values of the variables a,b and c, write a program to rotate their values such that a has the value of b, b has the value of c, and c has the value of a.
16. Write a program to print out the largest value of given three numbers without using if statement.
17. Explain about the formatted I/O Statement.
18. Explain about unformatted I/O statements.

UNIT - II

DECISION MAKING, ARRAYS AND STRINGS

OBJECTIVES

After reading this unit, the student will be able to

- Learn about decision making control statements in C and the way they are used.
- Learn about looping control constructs in C and the technique of putting them to use.
- Describe the use of unconditional statements with their syntax.
- Learn about nested loops and their utility.
- Understand what an array is
- Learn about one-dimensional arrays, their declaration, initialization, ways to access individual elements and other possible operations.
- Get acquainted with one dimensional string and the way it is declared, initialized, manipulated, inputted and displayed.
- Define two dimensional and multidimensional arrays
- Know about array of strings, its declaration, initialization, other operations, manipulations, and uses.
- Get a brief idea about string handling functions
- Write simple programs using decision making statements, arrays and strings

INTRODUCTION

Normally the statements will be executed sequentially. Control statements are used to specify the order in which the various instructions in a program are to be executed. Control statements are used to determine the flow of control in a program. The control statements used in C are grouped into three categories. 1. Decision making or Branching statements 2. Looping statements 3. Unconditional Statements. Decision making statements are used to execute particular set of instructions for a particular situation. Looping statements are used to execute a group of instructions repeatedly until some condition is satisfied. Unconditional statements are used to transfer the control to other statements without checking any condition.

Arrays are a kind of data structure that can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type .

In this unit , we will discuss about the different types of branching statements, looping statements , arrays and Strings.

2.1. DECISION MAKING AND BRANCHING STATEMENTS

Decision making in C

Decision making is about deciding the order of execution of statements based on certain conditions or repeat a group of statements until certain specified conditions are met. C language handles decision-making by supporting the following statements,

- a. *if* statement
- b. *switch* statement
- c. conditional operator statement
- d. *goto* statement

The *if* statement may be implemented in different forms depending on the complexity of conditions to be tested. The different forms are,

- Simple *if* statement
- *If....else* statement
- Nested *if....else* statement
- *else if* statement

2.1.1. Simple if Statement

Simple if statement is used to execute some statements for a particular condition. The general form of if statement and flow chart is shown below:

Syntax	Flow chart
<pre>if (condition) { statement-1; } statement -2;</pre>	<pre> graph TD entry((entry)) --> condition{condition} condition -- true --> statement1[statement-1] statement1 --> statement2[statement-2] condition -- false --> statement2 </pre>

where condition is a relational or a logical expression. The condition must be placed in parentheses. statement-1 can be either simple or compound statement (group of Statements). The value of condition is evaluated first. The value may be either true or false. If the condition is true, the statement – 1 is executed and then the control is transferred to statement – 2. If the condition is false, the control is transferred directly to the statement-2 without executing the statement-1.

The conditional statement should not be terminated with Semi-colons (ie ;)

Example 1: Write a program to check equivalence of two numbers. Use "if" statement

```
# include<stdio.h>
# include<conio.h>
void main()
{
int m,n; clrscr( );
printf("\n Enter two numbers:");
scanf("%d %d", &m, &n);
if(m==n)
printf("\n two numbers are equal");
getch(); }
```

Example 2: Write a Program to check whether a given number is less than 20.

```
#include <stdio.h>
# include<conio.h>
void main ()
{
int a;
scanf("%d", &a);
if( a < 20 )
{
printf("a is less than 20\n");
} printf("value of a is : %d\n", a); }
```

2.1.2. if else Statement

The simple if statement will execute a single statement, or a group of statements, if the condition is true. If the condition is false, it does nothing.

If ...else statement is used to execute one group of statements if the condition is true. The other groups of statements are executed when the condition is false. General form of if...else statement and flow chart is shown below:

Syntax	Flow chart	Explanation
<pre>if (condition) { statement-1; } else { statement-2; } statement -3;</pre>	<pre> graph TD entry((entry)) --> cond{Condition} cond -- true --> stat1[Statement-1] stat1 --> merge(()) cond -- false --> stat2[Statement-2] stat2 --> merge merge --> stat3[Statement-3] </pre>	<p>statement-1 and statement-2 can be simple or compound statements. If the condition is true, the statement-1 is executed; otherwise the statement-2 is executed. In either case, either statement-1 or statement-2 will be executed, not both. In both the cases, the control is transferred subsequently to statement-3.</p>

Example 1: Write a program to print the given number is even or odd.

```
# include<stdio.h>
void main( )
{ int n;
printf("Enter a number:");
scanf("%d", &n);
if( (n%2)==0 )
printf("\n The given number is EVEN");
else
printf("\n The given number is ODD");
}
```

Example2: Develop a program to accept two numbers and find largest number and print.

```
# include<stdio.h>
void main( )
{ int a,b;
printf("Enter Two numbers:");
scanf("%d%d", &a,&b);
if( a>b )
printf("\n %d is largest number",a);
else
printf("\n %d is largest number",b);
}
```

2.1.3. Nested ifs

A nested if statement is an if statement which is within another if – block or else – block. If an if...else is contained completely within another construct, then it is called nesting of if's.

Syntax	Flow chart
<pre>.if (condition-1) { if (condition-2) { statement-1; } else { statement-2; } else { statement -3; } } Statement – 4;</pre>	<pre> graph TD entry((entry)) --> cond1{Condition-1} cond1 -- true --> cond2{Condition-2} cond1 -- false --> stat3[Statement-3] cond2 -- true --> stat1[Statement -1] cond2 -- false --> stat2[Statement-2] stat1 --> merge(()) stat2 --> merge merge --> statX[Statement-X] </pre>

If the condition-1 is false, the statement-3 will be executed. Otherwise it continues to test condition-2. If the condition-2 is true, the statement-1 will be executed; otherwise the statement-2 will be executed and then the control is transferred to statement-4. Second if...else construction may be nested in the if block or else block of the first if...else construction. The statements between the keywords if and else are called if block and statements after the else form the else block.

Example: Program to select and print the largest of the three float numbers using nested “if-else” statements.

```
# include<stdio.h>
# include<conio.h>
void main( )
{
float a,b,c;
printf("Enter Three Values:");
scanf("%f%f%f", &a, &b, &c);
printf("\n Largest Value is:");
if(a>b)
{
    if(a>c)
        printf(" %f ", a);
    else
        printf(" %f ", c); }
else
{
    if (b>c)
        printf(" %f ", b);
    else
        printf(" %f ", c); }
getch( ); }
```

In the above example second if....else construct is used within the else block and if block of the first if...else construct.

2.1.4. Else if ladder

The else-if ladder is multiway branching statement. It is used to test many conditions. .

Syntax	Flow chart
<pre>if (condition 1) statement 1; else if (condition 2) statement 2; else if (condition 3) statement 3; else if (condition n) statement n; else default statement;</pre> <p style="text-align: right;">statement x;</p>	<pre> graph TD C1{condition 1} -- True --> S1[Statement 1] S1 --> C2{condition 2} C2 -- True --> S2[Statement 2] S2 --> C3{condition 3} C3 -- True --> S3[Statement 3] S3 --> Cn{condition n} Cn -- True --> Sn[Statement n] Cn -- False --> DS[Default statement] Sn --> Sx[Statement x] Sx --> NS[Next statement] </pre>

The conditions are evaluated from top to bottom. When a true condition is evaluated, the statement associated with it is executed and the rest of the ladder is omitted. If none of the conditions is true

then default statement is executed. If the default statement is not present, no action taken place when all other conditions are false.

Example

The marks obtained by a student in 5 different subjects are input through the keyboard. The student gets a division as per the following rules: (i) Percentage above or equal to 60 - First division (ii) Percentage between 50 and 59 - Second division (iii) Percentage between 40 and 49 - Third division (iv) Percentage less than 40 - Fail . Write a program to calculate the division obtained by the student.

```
main( )
{
    Int m1, m2, m3, m4, m5, per ;
    per = ( m1+ m2 + m3 + m4+ m5 ) / 5 ;
    if ( per >= 60 )
        printf ( "First division" ) ;
    else if ( per >= 50 )
        printf ( "Second division" ) ;
    else if ( per >= 40 )
        printf ( "Third division" ) ;
    else
        printf ( "fail" ) ;
}
```

If the first condition (per ≥ 60) is true, then “First division” is printed. and the control will come out of the loop. If the first condition is false, then the next statement (per ≥ 50) will be evaluated. Depending upon the result, the next statement is executed or control will come out of the loop. The above procedure is continued until the last else if.

2.1.5. Switch Statement

1. The *switch* statement causes a particular group of statements to be chosen from several available groups.
2. The selection is based upon the current value of an expression which is included within the *switch* statement.
3. The *switch* statement is a multi-way branch statement.
4. In a program if there is a possibility to make a choice from a number of options, this structure is useful.
5. The *switch* statement requires only one argument of *int* or *char* data type, which is checked with number of case options.
6. The *switch* statement evaluates expression and then looks for its value among the *case* constants.
7. If the value matches with *case* constant, then that particular *case* statement is executed.
8. If no one *case* constant not matched then *default* is executed.
9. Here *switch*, *case*, *break* and *default* are reserved words or keywords.
10. Every *case* statement terminates with colon “:”.
11. In *switch* each *case* block should end with *break* statement

```
switch (expression)
{
    case constant1 ;
        statement – 1 ;
}
```

```

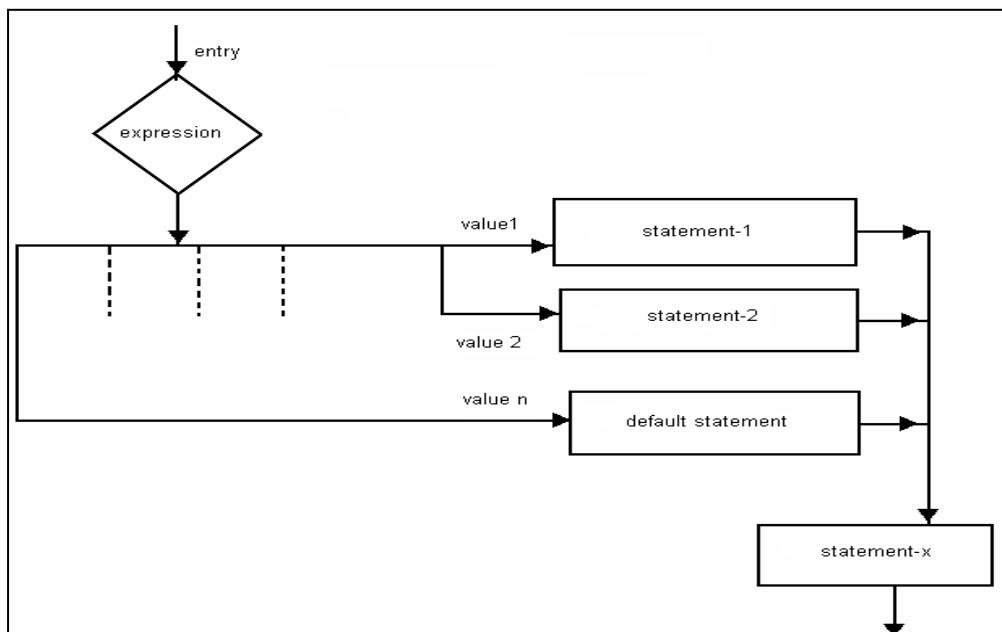
break;
case constant2 ;
    statement - 2 ;
    break;
default:
    default_statement ;
}
statement -x;

```

where statement-1, statement-2 are statement lists and may contain zero or more statements. The expression following switch must be enclosed in parentheses and the body of the switch must be enclosed within curly braces. Expression may be a variable or integer expression. Case labels can be only integer or character constants.

case labels do not contain any variable names. case labels must all be different . case labels end with a semicolon.

The switch structure starts with the switch keyword. It contains one block which contain the different cases. Each case contains different statements to be executed corresponding to different conditions and ends with the break statement. Break statement transfers the control out of the switch structure to statement – x. If the value of variable is “constant1”, the “case constant1: “is executed. If value is “constant2”, the “case constant2” is executed and so on. If the value of the variable does not correspond to any case, the default case is executed.



Flow chart for switch Statement

Example

```

void main ( )
{
    char ch;
    int a,b,c=0;
    printf ("Enter the two values");
    scanf("%d%d",&a, &b);
    printf ("enter the operator ( + - * / )");

```

```

ch = getchar();
switch(ch)
{
    case "+": c = a + b;
                break;
    case "-": c = a - b;
                break;
    case "*": c = a*b;
                break;
    case "/": c = a/b;
                break;
    default: printf ("The operator is invalid");}
printf ("The result is %d", c);
}

```

RULES FOR FORMING SWITCH STATEMENT

1. The order of the *case* may be in any order. It is not necessary that order may be in ascending or descending.
2. Mixing of integer and character constants in different cases is allowed.
3. If multiple statements are to be executed in each case, there is no need to enclose within a pair of braces.
4. If default statement is not present, then the program continues with the next instruction.
5. switch statement may occur within another switch statement.
6. If the break is not used in a certain case, the statements in the following cases are also executed irrespective of whether that case value is true or not. This execution will continue till a break is encountered. For example, consider the following program,

```

switch( ch )
{
    case "a":
    case "b":
    case "i": printf ("The vowel i \n");
    case "o": printf ("The vowel o");
                break ;
    case "u":
    default : printf ( Not vowel );
}

```

7. In the above example, if the value of ch = 'i', then the display will be

The vowel i
The vowel o
8. When one of the case statements is evaluated as true, all statements are executed until a break statement is executed.

2.1.6. GOTO Statement

The goto statement is used to transfer the control from one point to another. The general form of the goto statement is

goto label;

where label is an identifier. Label is the name given to the target place to which the control will be transferred. Control may be transferred to any other statement within the program. The label is

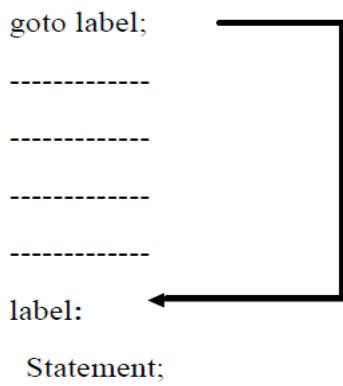
placed immediately before the statement where the control is to be transferred. The label can be anywhere in the program either before or after the goto label statement. The general form of the target place is

Label: statement (s)

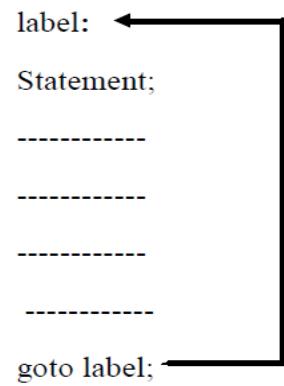
The goto statement transfers the control without checking any condition. That is why this statement is sometimes called as unconditional goto statement. *Goto statement may be useful for exiting from any levels of nesting in one jump.*

It is possible to have a forward jump or a backward jump.

- If the “label:” is before the statement “goto label;” a loop will be formed and some statements will be executed repeatedly. Such a jump is known as a „backward jump”.
- If the “label:” is placed after the “goto label;” some statements will be skipped and the jump is known as a “forward jump”.



Forward jump



Backward jump

Example

```
main ( )
{
    int x = 10;
    loop:
        printf ("x is %d \n", x );
        x++;
        if ( x < 100 ) goto loop;
        printf ("End of the goto statement");
}
```

2.2. LOOPING STATEMENTS

Loops are used to execute a same set of instructions for many times. The number of times a loop executed is given by some condition. The loop structures available in C are

1) for loop 2) while loop and 3) do...while loop

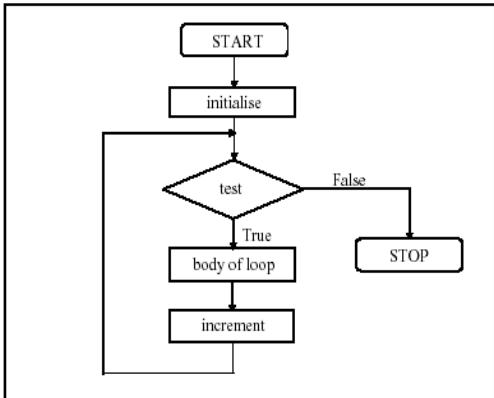
2.2.1. for loop

The for loop is used to repeat a statement or block of statements for a known number of times. The general format of the for loop is

```

for ( initialization; condition test; increment)
{
    statements(s)
}

```



Initialization is an assignment statement, that is used to set the loop control variable. The condition test is a relational or logical expression which determines when to end the loop. The increment is a unary or an assignment expression. This section is used to alter the value of the variable initially assigned by initialization. These three sections must be separated by semicolon. The statement which forms the body of the loop can be either a single statement or a compound statement (group of statements).

When the “for statement” is executed, the value of conditional test is evaluated and tested before each pass through the loop. Incrementation is carried out at the end of each pass.

The for loop continues to execute as long as the value of the conditional test is true. When the value of condition becomes false, the program comes out of the for loop.

Execution of the for loop

1. Initialization of control variable is done first.
2. The value of control variable is tested using condition test. If the value is true, the body of the loop is executed; otherwise the loop is terminated.
3. After the body of the loop is executed, the control is transferred back to for statement. Control variable is altered and now the new value is tested. This process continues till the value of the control variable fails to satisfy the test condition.

Example: To print odd numbers from 1 to 13

```

main()
{
    int i;
    for ( i=1; i<=13; i=i+2)
        printf("%d", i);
}

```

Output : 1 3 5 7 9 11 13

The above for loop initialized the integer variable i to 1 and increments it by 2 every time the loop is executed.

In for loop, the conditional test is always performed at the starting of the loop. The body of the loop is not executed if the condition is false in the beginning itself. **Thus the minimum number of loop execution is zero.**

All the three sections need not be present in the for statement. But semicolons are necessary and must be shown. If the first section is omitted, the initialization is to be done before the for loop. If the third section is omitted, the variable is incremented in the body of the loop. If the second section is omitted, it will be assumed that the permanent value is 1; then the loop will continue indefinitely.

Different forms of for loop

Different forms of statements to print the numbers from 1 to 10 are given below.

1. The initialization, testing and incrementation of the control variable is in the for statement itself.

```
main ( )  
{           int i;  
          for ( i = 1; i <= 10; i = i + 1 )  
          printf ( "%d",i );  
}
```

2. Incrementing the control variable in the body of the for loop.

```
main ( )  
{  
    int i;  
    for ( i = 1; i <= 10; )  
    {  
        printf ('%d', i );  
        i = i + 1;  
    }  
}
```

3. Initialization in the declaration statement itself, but before the condition..

```
main ()  
{     int i = 1;  
      for ( ; i <= 10 ; i = i + 1 )  
      printf ('%d', i );  
}
```

4. Initialization is in declaration statements and incrementation is in the body of the loop.

```
main()  
{     int i = 1;  
  for ( ;i <= 10;)  
  {  
      printf ('%d', i);  
      i = i + 1;  
  }  
}
```

5. Comparison and incrementation is made through the same statement

```
main()  
{     int i ;  
  for ( i = 0; i++ < 10; )  
  printf (" %d ", i );  
}
```

Multiple Initializations and Increments in for loop

The initialization section and increment section can contain more than one variable; These variables are separated by the operator comma and evaluated from left to right.

Example

```
for ( i = 0, j = 10; i < 50; i++, j-- )  
{  
    body of the loop  
}
```

If the multiple expressions are to be checked, then they are connected by logical operators.

Infinite loop

A loop becomes **infinite** loop if a condition never becomes **false**. The **for loop** is traditionally used for this purpose. Endless loop can be formed by leaving the conditional expression empty. Infinite loop can be formed in two ways. They are

By writing no condition. For example - `for (i = 1; ;i++)`

By omitting all the sections of a for statement - `for (;;)`

When the conditional expression is absent, it is assumed to be true. You may have an initialization and increment expression, but C programmers more commonly use the `for(;;)` construct to signify an infinite loop.

Time Delay Loop

For loop can also be used without the body of the loop. This type of loop create time delays

```
for (i = 0; i<=1000; i++ );
```

is an example of a time delay loop. A for statement with a semicolon at the end of the statement is called null statement.

2.2.2. while loop

The while loop construct contains only the condition. The general format of the while loop is

<i>Initialization Expression;</i> <i>while(Test Condition)</i> { Body of the loop <i>Updation Expression }</i>
--

where body of the loop is either an empty statement, a single statement or a block of statements. The condition may be any expression. The condition value may be zero or non-zero.

1. The **while** is an entry – controlled loop statement.
2. The test condition is evaluated and if the condition is true, then the body of the loop is executed.
3. The execution process is repeated until the test condition becomes false and the control is transferred out of the loop.
4. On exit, the program continues with the statement immediately after the body of the loop.
5. The body of the loop may have one or more statements.
6. The braces are needed only if the body contains two or more statements.
7. It is a good practice to use braces even if the body has only one statement.

Example

The following example prints the value from 1 to 100

```

{
    int i = 1 ;
    while ( i<=100 )
    {
        printf ("%d\n",i);
        i++;
    }
}

```

Like for loop, while loop checks the condition at the top of the loop. So the body of the loop is not executed, if the condition is false at the starting of the loop.

The for loop can be used if the number of iterations are known before the loop starts executing. When the number of iterations is not known, then while loop can be used.

2.2.3. do.....while loop

The do while loop is sometimes called as the do loop in C. Unlike for and while loops, this loop checks the condition at the end of the loop. **So the body of the loop is executed at least once, even if the condition is false initially.** The general form of do...while loop is

<i>Initialization Expression;</i> <i>do</i> { Body of the loop <i>Updation Expression; }</i> <i>while (Test Condition);</i>

The do...while loop repeats until the condition becomes false. In the do...while, the body of the loop is executed first, then the condition is checked. When the condition becomes false, control is transferred to the statement after the loop.

Example :

To print the values from 1 to 10

```

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

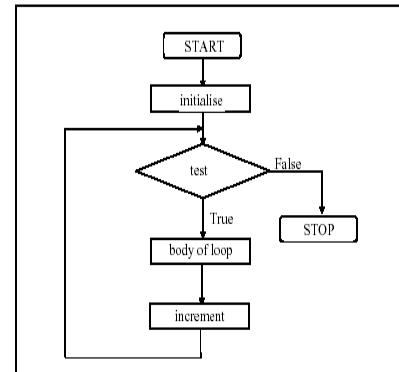
```

Difference between while loop and do-while loop

The difference between the while and do...while is illustrated by the following program segments

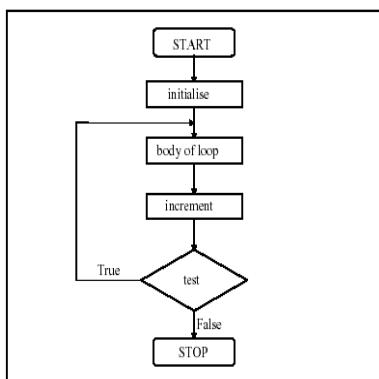
while loop

```
main ( )
{
    while ( 100 < 10 )
        printf ("False");
}
```



In the above program, the condition is false for the first line. So the printf () will not be executed at all.

do...while loop



```
main ( )
{
    do
    {
        printf ("False");
    } while (100 < 10 );
}
```

In the above program, the printf statement will be executed only once.

Nested loops

A loop can also be used within loops.

1. i.e. one *for* statement within another *for* statement is allowed in C. (or „C“ allows multiple *for* loops in the nested forms).
2. In nested *for* loops one or more *for* statements are included in the body of the loop.

Two loops can be nested as follows.

Syntax:

```
for( initialize ; test condition ; increment) /* outer loop */
{
    for(initialize ; test condition ; increment) /* inner loop */
        {Body of loop; }
}
```

For example, to find the factorial value of number between 1 to 10, the program is

```

main ( )
{
    int i, j, factorial;
    for ( i=1; i<=10; i++ )
    {
        factorial = 1;
        for ( j = 1; j<= i; j ++ )
        {factorial = factorial * j;}
        printf ("Factorial value of %d is %d", i, factorial );
    }
}

```

One loop can be completely contained within the other. But there can be no overlap. Each loop must have different control variable.

counter controlled and sentinel controlled loops

Based on the nature of the control variables and the kind of value assigned to, the loops may be classified into two general categories; counter controlled and sentinel controlled loops.

Counter controlled loops : The type of loops, where the number of execution is known in advance are termed by the counter controlled loop. That means, in this case, the value of the variable which controls the execution of the loop is previously known. The control variable is known as counter. A counter controlled loop is also called definite repetition loop.

Example : A while loop is an example of counter controlled loop.

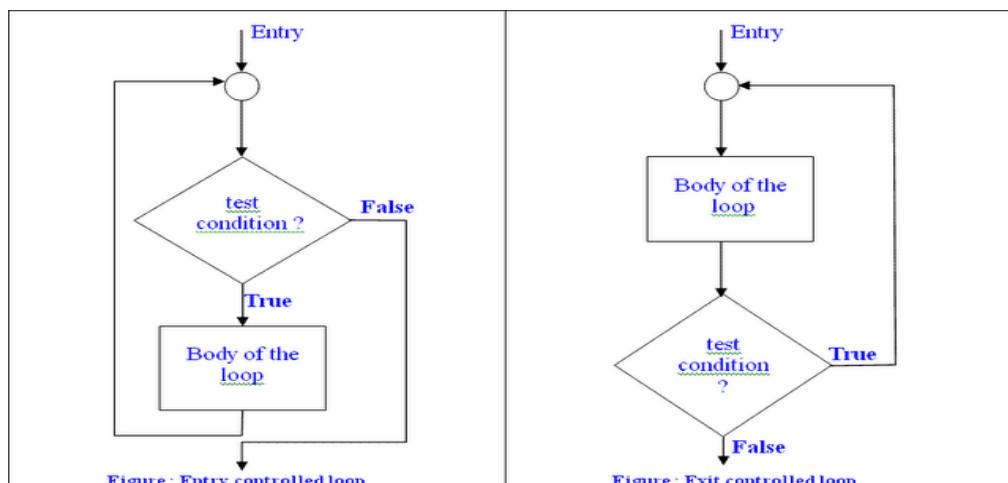
Sentinel controlled loop : The type of loop where the number of execution of the loop is unknown, is termed by sentinel controlled loop. In this case, the value of the control variable differs within a limitation and the execution can be terminated at any moment as the value of the variable is not controlled by the loop. The control variable in this case is termed by sentinel variable.

Example : do....while loop is an example of sentinel controlled loop.

entry controlled and exit controlled loops

Depending on the position of the control statement in the loop, a control structure can be classified into two types; entry controlled and exit controlled. They are described below.

Entry controlled loop : The types of loop where the test condition is stated before the body of the loop, are known as the entry controlled loop. So in the case of an entry controlled loop, the condition is tested before the execution of the loop.



If the test condition is true, then the loop gets the execution, otherwise not. For example, the **for loop** is an entry controlled loop. In the given figure, the structure of an entry controlled loop is shown.

Exit controlled loop : The types of loop where the test condition is stated at the end of the body of the loop, are known as the exit controlled loops. So, in the case of the exit controlled loops, the body of the loop gets execution without testing the given condition for the first time. Then the condition is tested. If it comes true, then the loop gets another execution and continues till the result of the test condition is not false. For example, the **do....while loop** is an exit controlled loop. The structure of an exit controlled loop is given in the given figure.

Comparison between the three loops, for, while, do....while.

No.	Topics	For loop	While loop	Do...while loop
01	Initialization of condition variable	In the parenthesis of the loop.	Before the loop.	Before the loop or in the body of the loop.
02	Test condition	Before the body of the loop.	Before the body of the loop.	After the body of the loop.
03	Updating the condition variable	After the first execution.	After the first execution.	After the first execution.
04	Type	Entry controlled loop.	Entry controlled loop.	Exit controlled loop.
05	Loop variable	Counter.	Counter.	Sentinel & counter

2.2.4. Break Statement

1. A break statement terminates the execution of the loop and the control is transferred to the statement immediately following the loop.
2. The break statement is used to terminate loops or to exit from a switch.
3. It can be used within a for, while, do-while, or switch statement.
4. The break statement is written simply as break;

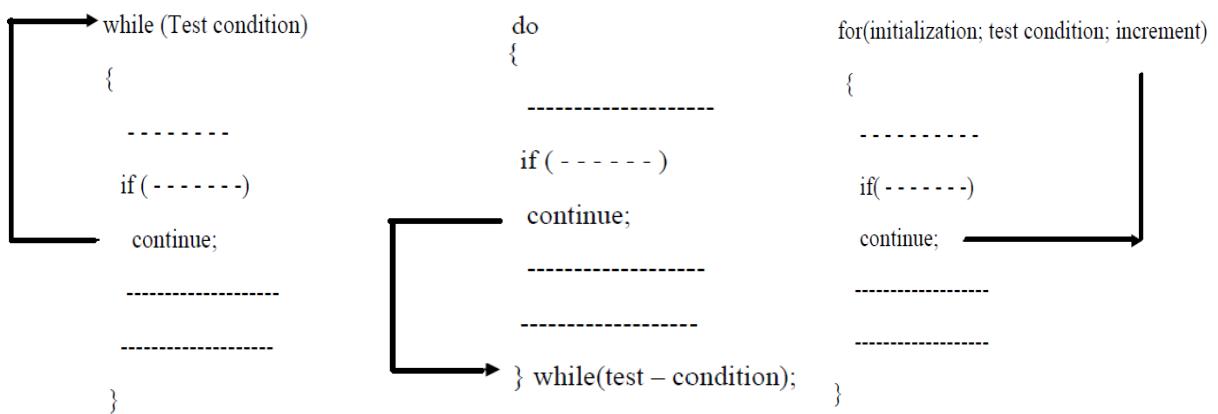
Example

```
main()
{
    int ch = 65;
    for(;;ch++)
    {
        printf("%c", ch);
        if (ch == 97) break;
    }
}
```

2.2.5. Continue Statement

1. The continue statement is used to bypass the remainder of the current pass through a loop.
2. The loop does not terminate when a continue statement is encountered.

3. Instead, the remaining loop statements are skipped and the computation proceeds directly to the next pass through the loop.
 4. The continue statement can be included within a *while*, a *do-while*, a *for* statement.
 5. It is simply written as “continue”.
 6. The continue statement tells the compiler “Skip the following Statements and continue with the next Iteration”.
 7. In “while” and “do” loops continue causes the control to go directly to the test – condition and then to continue the iteration process.
 8. In the case of “or” loop, the updation section of the loop is executed before test-condition, is evaluated.



Example

```
main()
{
    int number, sum=0, count=0;
    while (count <= 10)
    {
        printf("\nEnter the number: ");
        scanf("%d",&number);
        if(number <=0 ) continue;
        sum += number;
        count++;
    }
}
```

The above program is used to find the sum of positive numbers.

2.3. ARRAYS

2.3.1. Introduction

An array contains a collection of data elements of the same type. An array is referenced by a common name. Each element of an array is stored in successive locations of the memory.

Types of Arrays: Arrays can be used to represent not only simple lists of values but also tables of data in two or three or more dimensions.

- a. One – dimensional arrays
 - b. Two – dimensional arrays
 - c. Multidimensional arrays

2.3.2. Array elements and Subscript

The elements of the array are known as *members of the array*. Each array element is identified by assigning a unique subscript or index to it. The dimension of an array is determined by the number of subscripts needed to identify each element. A subscript is enclosed in bracket [] placed after the array name. Space is not allowed between array name and subscript. A subscript is an integer value starting from zero.

Thus the array named mark with 5 elements will be represented as mark[0], mark [1], mark[2], mark [3], mark [4], and stored in successive locations of the memory as shown in the following figure.

mark[0]	mark[1]	mark[2]	mark[3]	mark[4]
---------	---------	---------	---------	---------

Subscripts always start with 0. So, for an array of N elements, the last element has an index of N-1.

2.3.3. One dimensional array declarations

In one-dimensional array, single subscript is used. *The one-dimensional array is also called as list*.

Before using an array, it must be declared. An array is defined in the same way as a variable. Each array name must be accompanied by a size specification (i.e. number of elements). The general form for array declaration is

storage-type data-type array_name[SIZE]

where *storage-type* is optional. It may be either static, extern, automatic. The *data-type* specifies the type of element that will be stored in the array. *SIZE* is an integer constant, indicating the maximum number of elements of the array. The rules for forming array names are the same as for variable names.

Examples

(i) int number[100]; (ii) char text[100]; (iii) float ave[10]; (iv) signed char name[50];

In the first example, *int* specifies the type of the variable and the word *number* specifies the name of the array. The number *100* indicates the maximum number of elements in the array. This number is called as ‘dimension’ of the array. The bracket ([]) indicates that the given variable is an array.

The size of array should be a constant value.

The main purpose of the declaration of the array is to reserve space in memory. For example when declaring the array

```
int marks[10];
```

reserves the $10 \times 2 = 20$ bytes in memory. Each int occupies 2 bytes of memory.

2.3.4. Initialization of Arrays

After an array is declared, its elements must be initialized. Otherwise they will contain “garbage”.

An array can be initialized at either of the following stages. (i) At compile time (During declaration of the array) (ii) At run time. (using for loop and scanf() functions)

1. During the declaration of the array

The general form of declaring the array during the initialization is

```
data_type array_name[SIZE] = {list of values}
```

The values in the list must be separated by commas.

Examples :

```
int mark[5] = {55,60,70,23,100};      float avg[3] = {25.6,75.5,23.3};
```

In the first example, the size of the array is 5 and the values 55, 60, 70, 23, 100 will be assigned to the variable mark[0], mark[1], mark[2], mark[3] and mark[4] respectively.

If the number of values in the list is less than the size of the array, the remaining elements will be set to zero automatically.

For example, the statement **float height[4] = {0.25, 0.50};** will initialize the first two elements 0.25 and 0.50 to height[0] and height[1] respectively. The remaining two elements height[2] and height[3] will be set to 0.0

The size of the array may be omitted. In this situation, the compiler allocates enough space for all the elements given for initialization. For example, the statement **int marks[] = {25, 50, 100, 75};** will declare the size of the array is 4.

If there are more elements than the declared size, the compiler will produce an error.

int number[3] = {10,20,30,40}; will not work. It is illegal in C.

2. Initialization of an array using for...loop.

for...loop is used to initialize values for array elements. for...loop can be used when initializing the constant value or values which are having some relation to the array which are having large number of elements.

Examples

(i) int array[100], i ;
 for (i = 0; i < 100; i++)
 array [i] = 0;

The above for...loop initialize all the elements with the value of 0.

(ii) int array[50], i, j = 0;
 for (i = 0; i < 50; i++)
 {
 array [i] = j;
 j = j + 2;
 }

The above for...loop initializes the value 0 to array[0], 2 to array[1], 4 to array[2] and so on.

(iii) To initialize different values, **scanf()** function is used in the body of the for-loop statement.

```
int array[50], i ;  
for ( i = 0 ; i <= 50; i++ )  
scanf("%d", &array[i]);
```

2.3.5. Two dimensional Arrays

One-dimensional arrays have a single subscript. But two-dimensional arrays have two subscripts. The two dimensional array is called a *matrix or table*. Two dimensional arrays are used to represent the values which are in matrix form. A two dimensional array can be represented with two pairs of square bracket.

Two-dimensional array can be represented in the following form

data-type array_name[subscript1][subscript2]

where subscript1, subscript2 are positive value integer constants or expression. This two subscripts indicates the number of array elements associated with each subscript.

Two square brackets are used to indicate the value of two subscripts. Assume the values of two subscripts as 'm' and 'n'. Then two dimensional array can be arranged as a table with m rows and n columns.

Examples : float table [3] [3]; int marks [10] [5];

The first example defines table as a floating-point array having 3 rows and 3 columns. An array element starts with an index of 0 so that the individual elements of the array are

table	[0][0]	[0][1]	[0][2]
	[1][0]	[1][1]	[1][2]
	[2][0]	[2][1]	[2][2]

The total number of elements of the two dimensional array can be calculated by multiplying the number of rows and number of columns.

Example for a two - dimensional array

A typical example for a two – dimensional array is the marks obtained by 50 students in 5 tests. To represent the above values 50*5 matrix is used. This can be declared by

int marks [50] [5];

The first index ([50]) is the number of the students and the second index ([5]) is the number of tests. This declaration allocates 500 memory locations. The test number is the column number and the student number is the row number. In each row, the marks obtained by the student in all the 5 tests are represented.

	Test1	Test2	Test3	Test4	Test5
student1	90	96	98	100	99

student50	79	87	88	67	55

INITIALIZATION OF A TWO DIMENSIONAL ARRAY

Care must be given to the order in which the initial values are assigned to the array elements. The second subscript increases most rapidly and the first subscript increases least rapidly. The elements of a two dimensional array will be assigned row wise. That is, all the elements of the first row will be assigned, then all the elements of the second row and so on. For example, consider the following two-dimensional array mark [4] [3] (three test marks of four students)

```
    75  25  30  
mark [4] [3] = 45  50  22  
              40  72  45  
              41  55  78
```

The above example array can be initialized as: int mark [4] [3] = {75,25,30,45,50,22,40,72,45,41,55,78};

The first subscript ranges from 0 to 3 and the second subscript ranges from 0 to 2. Array elements will be stored in continuous locations in memory. Two dimensional arrays are also can be initialized in the following way

```
int mark [4] [3] = {  
    {75, 25, 30},  
    {45, 50, 22},  
    {40, 72, 45},  
    {41, 55, 78},  
};
```

The three values in the first inner pair of braces are assigned to the array elements in the first row, the values in the second inner pair of braces are assigned to the array elements in the second row and so on. An outer pair of braces is required, containing the inner pairs. Each line contains the three marks of one student separated by comma and enclosed in braces and separated from the next students' marks by comma. The whole array is enclosed in a pair of braces.

While initializing an array, the second dimension is a must. The first dimension (row) is optional. Thus the following two declarations

```
int mark [2] [3] = {12, 24, 36, 48, 60, 70};  
int mark [ ] [3] = {12, 24, 36, 48, 60, 70};
```

are equal.

If the values are missing in an initialization, they are automatically set to zero. For example

```
int mark [4] [3] = {  
    {12, 75},  
    {55, 75, 23},  
    {45, 98, 57},  
    {48}  
};
```

will assign the following values

```

mark[0][0] = 12   mark[0][1] = 75   mark[0][2] = 0
mark[1][0] = 55   mark[1][1] = 75   mark[1][2] = 23
mark[2][0] = 45   mark[2][1] = 98   mark[2][2] = 57
mark[3][0] = 48   mark[3][1] = 0    mark[3][2] = 0

```

Initialization by using for statement

The initialization can be done by using two for-loop statements. Here all the values are initialized to zero.

```

for ( i = 0; i < 4; i++ )
{
    for ( j = 0; j < 3; j++ )
    {
        mark[i][j] = 0;
    }
}

```

The for statement is used when initializing the same value or values which are having some relationship to each other to array elements.

Two dimensional arrays are used to initialize a set of string values. For example to store the name of 10 students (each name with a maximum of 30 characters) in a class, the following code is used

```

for ( i = 0; i < 10; i++ )
{
    for ( j = 0; j < 30; j++ )
    {
        scanf("%s", name[i]);
    }
}

```

2.3.6. Multi Dimensional Arrays

An array of three or more dimension is called as multidimensional array. The general form of a multi-dimensional array is

data-type array_name[subscript1][subscript2]][subscript2]

where subscript 1 is size of the i-th dimension.

Examples

```

float sales[4][5][12];
int table[2][4][10][5];

```

where sales is a three-dimensional array, contains 240 float type elements. Similarly table is a four dimensional array containing 400 elements of int data type. The array sales may represent a data of sales during the last four years from January to December in five cities of a particular company.

Multi dimensional array can contain as many indices as needed. However the amount of memory needed for an array rapidly increases with each dimension. For example

```

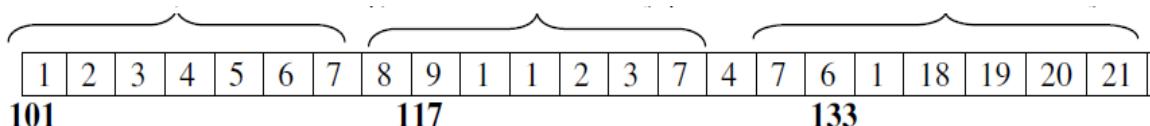
char arr [100][365][24][60][60]; declaration would consume more than 3 gigabytes of memory.

```

Memory does not contain rows and columns, so whether it is a one dimensional array or two dimensional arrays, the array elements are stored linearly in one continuous chain. For example, the multidimensional array

```
int arr[3][4][2]= {  
    { {1,2},{3,4},{5,6},{7,8} },  
    { {9,1},{1,2},{3,7},{4,7} },  
    { {6,1},{18,19},{20,21} },  
};
```

is stored in memory just like an one-dimensional array shown below:



2.4 CHARACTER STRINGS

2.4.1. Introduction

A group of characters defined between double quotation marks is called as a string constant. A group of characters can be stored in a character array. Character arrays are called as “strings”.

Whenever a string is stored in memory, the NULL character ('\0') is automatically added to its end. This Null character indicates the end of the string. A string not terminated by a '\0' is not really a string, but merely a collection of characters.

2.4.2. Declaration and Initialization of string

Strings are declared by using the data-type **char**. The general form for declaring the string is

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

The size determines the number of characters in the array-name.

Example: (i) char name[10]; (ii) char place[20];

In the second example, the character array ‘place’ can store 19 characters. place [0] to place [18] is for storing valid characters and place [19] is used for storing the NULL character ‘\0’. When declaring character arrays, always allow one extra element space for the null character.

When a string is read by using any of the string-based input functions, namely scanf() and gets(), the NULL character is automatically appended to the string.

Initialization of String

Initialization of character array is possible during the declaration. The strings are initialized in the following ways:

Method 1 : char name[7] = {"t", "e", "m", "p", "l", "e", "\0"};

Method 2 : char name[7] = {"temple"};

When using the method 2, the declaration '\0' is not necessary. C inserts the NULL character automatically. But when initializing character array by listing its elements, NULL termination must be specified at the end.

2.4.3. Reading Strings from Terminals

There are two ways to read a string from the terminal. They are (1) by using the scanf function and (2) by using gets function.

(i) By using scanf function

The scanf function with %s format specification can be used to read a string of characters. For example

```
char name[30];
scanf ("%s", name);
```

is used to input a string of 30 characters.

The disadvantage of scanf function is that there is no way to enter a multi-word string into a single variable. The scanf statement terminates its input on the first blank space it finds. Therefore if the input for the above statement is

```
SENTHIL KUMAR
```

then only the string SENTHIL will be read into the array name, since the blank space after the word 'SENTHIL' will terminate the string. During the reading of strings, in scanf statement, the ampersand (&) is not required before the variable name. The scanf function automatically terminates the string with a NULL character.

(ii) By using gets

gets() function is also used to read a string from the keyboard. It eliminates the disadvantages of scanf function. It is terminated when an enter key is pressed. The spaces and tabs are acceptable as part of the input string. The program can be written in the following way

```
main ()
{
    char name[30];
    print ("Enter a name \n");
    gets (name);
}
```

The above program is used to accept a string, which contains blank spaces and tabs. The value entered upto new line character or till the enter key is pressed is stored in the array name. If the input is

```
SENTHIL KUMAR
```

then the whole string SENTHIL KUMAR is stored in the variable name. *The disadvantage of gets is, it can be used to read only one string at a time.* For example, the statement

```
gets (name, address)
```

is not possible. It is possible to read more than one string by using a single scanf statement.

2.4.4. Writing of Strings

(I) By using printf () function

printf function with %s format is used to print strings on the screen. The format specifier %s is used to display a character array that is terminated by the NULL character. For example, the statement

```
printf ("%s", name);
```

can be used to display the entire contents of the array called "name".

(ii) By using puts () function

puts function is also used to print string on the screen. The general format of the puts function is

```
puts (variable name or string );
```

Unlike printf, puts() can output only one string at a time. When attempting to print two strings using puts (), only the first one will be printed.

Example

```
void main()
{
char name[30];
puts("Enter name");
gets(name);
puts(name);
puts("Good Morning")
}
```

OUTPUT:
Enter name
SURESH KUMAR
SURESH KUMAR
GOOD MORNING

2.4.5. Comparison of two strings

It is not possible to compare the two strings by using == operator. For example (if (str1 == str2) is not a valid statement in 'C'. To compare two strings, the two strings are compared character by character. The comparison is done until there is a difference or one of the strings terminates with a null character, whichever occurs first. The following program is used to compare two strings.

Example

```
void main()
{
    char name1[15], name2[15];
    int i = 0;
    printf ("\\n Enter the First string:");
    scanf("%s", name1);
    printf ("\\n Enter the Second string:");
    scanf("%s", name2);
    while(name1[i] == name2[i] && name1[i]!= "\\0" && name2[i] != "\\0")
        i++;
    if (name1[i] == "\\0" && name2[i] == "\\0")
        printf ("The two strings are equal");
    else
        printf( " The two strings are not equal");
}
```

2.4.6. String handling functions

The C library has large number of string-handling functions. These functions can be used to carry out many of the string manipulations. The most commonly used string-handling functions are shown below:

Function	Use	Function	Use
strcat()	To concatenate two strings	strcpy()	To copy one string over another string
strcmp()	To compare two strings	strlen()	To find the length of the string

strcat() function

The strcat function is used to join two strings. The general form of strcat() function is

```
strcat(string1,string2);
```

where string1 is a string variable and string2 may be a string variable or a string constant. When the function strcat is executed, string2 is appended to string1. strcat() removing the NULL character at the end of string1 and placing string2 from there. The string at string2 remains unchanged. For example, consider the following strings:

name1	S	I	V	A	\0			
name2	R	A	M	\0				

Execution of the statement

```
strcat(name1,name2);
```

will result in

name1	S	I	V	A	R	A	M	\0
name2	R	A	M	\0				

Make sure that the size of the string1 (to which string2 is appended) is large enough to accommodate the final string.

strcat function may also append a string constant to a string variable. The following is valid:

```
strcat (name, "SEKARAN");
```

C permits nesting of strcat functions. For example, the statement

```
strcat (strcat (string1, string2), string3);
```

is allowed and concatenates all the three strings together. The resultant string is stored in string1.

strcmp() function

It is not possible directly to compare the value of 2 strings in a condition like if(string1==string2) . strcmp function compares two strings character by character(ASCII comparisons) and returns a value. Value may be less than 0, or equal to 0, or greater than 0. The general structure of strcmp () function is

```
strcmp(string1,string2);
```

where string1 and string2 are the two strings to be compared. string1 and string2 may be either a string constant or a string variable. The following example illustrates the use of strcmp function.

Example:

1. `strcmp("Newyork","Newyork")` will return zero because 2 strings are equal.
2. `strcmp("their","there")` will return a 9 which is the numeric difference between ASCII 'i' and ASCII 'r'.
3. `strcmp("The", "the")` will return -32 which is the numeric difference between ASCII "T" & ASCII "t".

strcpy() function

`strcpy()` copies the second string to the first string specified in the `strcpy ()` parameter. The general form is

`strcpy(string1,string2);`

where `string1` is destination, `string2` is source, where `string1` is an array variable and `string2` is an array or a string constant.

Examples

1. `strcpy (str1, ""ABCD");` will copy the string 'ABCD' to the array `str1`.
2. `strcpy (name, name1);` will copy the contents of the array `name1` to the array `name`.

strlen() function

`strlen()` function is used to find the number of characters in string.

`n = strlen (string);`

where `n` is an integer variable which receives the value of the length of the string. The argument may be a string constant or a string variable. The counting ends at the first NULL character. NULL character is not included for counting.

atoi() function:

`atoi()` function is a C library function which is used to convert a string of digits to the integer value.

```
char st[10] = ""24175"" ;  
int n;  
n = atoi(st);
```

This will assign the integer value 24175 to the integer variable `n`.

Additional String Handling Functions:

- (i) **strupr()**: to convert all alphabets in a string to upper case letters.

Ex: `strupr("delhi") = " DELHI"`

(ii) strlwr(): To convert all alphabets in a string to lower case letters.

Ex: strlwr("CITY") = "city"

(iii) strrev(): To reverse a string

Ex: strrev("SACHIN") = "NIHCAS"

(iv) strncmp(): To compare first n characters of two strings.

x: m = strncmp(DELHI , DIDAR . 2); m = -4

(v) strcmpi(): To compare two strings with case in sensitive (neglecting upper / lower case)

Ex: m=strcmpi("DELHI ", "delhi "); m = 0.

(vi) strcat(): To join specific number of letters to another string.

Ex. char s1[10] = "New"; char s2[10] = "Delhi -41"; strcat(s1,s2,3); s1 will be "

NewDel".

PROGRAMS USING DECISION MAKING STATEMENTS, ARRAYS AND STRINGS

DECISION MAKING AND LOOPING

1. Find the output of the following program:

```
main()
{
    int i ;
    printf ( "Enter value of i " );
    scanf ( "%d", &i );
    if ( i = 5 )
        printf ( "You entered 5" );
    else
        printf ( "You entered something other than 5" ); }
```

Output : (Here is the output of two runs of the above program)..

Enter value of i 200

You entered 5

Enter value of i 9999

You entered 5

We have used the assignment operator = instead of the relational operator ==. As a result, the condition gets reduced to if (5), irrespective of the value of i. And remember that in C 'truth' is always nonzero, whereas 'falsity' is always zero. Therefore, if (5) always evaluates to true and hence the result.

2. Write a C Program that inputs an integer and determine whether it is evenly divisible by 5 and 7.

```
void main()
{
    int number;
```

```

printf("Enter the number:");
scanf("%d",&number);
if((number%5==0) &&(number %7==0))
    printf("The number is evenly divisible by 5 and 7");
else
    printf("The number is not evenly divisible by 5 and 7");
}

```

- 3.** In a Company an employee is paid as under: if his basic salary is less than Rs. 1500, then HRA = 10% of basic salary and DA = 90% of basic salary. If his salary is either equal to or above Rs. 1500, then HRA = Rs. 500 and DA = 98% of basic salary. If the employee's salary is input through the keyboard write a program to find his gross salary.

```

/* Calculation of gross salary */
void main( )
{
float bs, gs, da, hra ;
printf ( "Enter basic salary " ) ;
scanf ( "%f", &bs ) ;
if ( bs < 1500 )
{ hra = bs * 10 / 100.0 ;
da = bs * 90 / 100.0 ;}
else
{hra = 500 ;
da = bs * 98 / 100.- ;}
gs = bs + hra + da ;
printf ( "gross salary = Rs. %f", gs ) ;
}

```

- 4.** Write a C program to determine the factorial of a given number.

```

void main()
{
int number,fact=1,i;
printf("Enter the number:");
scanf("%d",number);
for(i=1;i<=number;i++)
    fact*=i;
printf("\n The factorial of given number is:%d",fact);
}

```

- 5.** Input three positive integers representing the sides of a triangle and determine whether they form a triangle. For a triangle, the sum of any two sides must be greater than the third side.

```

#include<stdio.h>
main()

```

```

{
int x,y,z;
clrscr();
printf("Enter the value for x ,y z:");
scanf("%d%d%d",&x&y&z);
if((x+y>z)&&(y+z>x)&&(x+z>y))
    printf("\n\n They form a triangle");
else
    pritnf("They does not form a triangle");
}

```

6. Write a program to solve the given quadratic equation $ax^2+bx+c=0$

```

void main()

{
float a,b,c,d;
double x1,x2;
printf("Enter the values of a,b and c \n");
scanf("%f %f %f",&a&b &c);
d=b*b-4*a*c;
if(d>0)
    {   printf("roots are real and different");
        x1=(-b+sqrt(d))/(2*a);
        x2=(-b-sqrt(d))/(2*a);
    }
else if(d==0)
    {           printf("roots are real and equal");
        x1=-b/(2*a);
        x2=x1;
    }
else
    printf("roots are real and imaginary");
printf("\n x1=%2f",x1);
printf("\nx2=%2f",x2);
}

```

7. Write a program to find the power of some number.

```

void main ( )
{
float x, power=1.0;
int n;
printf (" Type a number \n");
scanf ("% f",&x);
printf (" Raise this number to\n");
scanf ("%d", &n);

```

```

for( i=1; i<=n; i++)
{
power= power*x;
}
printf ("%f raised to %d is %f", x,n,power);
}

```

8. Using for loop, write a C program to find the sum of first 50 natural numbers.

```

void main()
{
int i,sum=0;
for(i=1;i<=50;i++)
    sum = sum + i;
printf("The sum is:%d",sum);
}

```

9. Write a program in C, which finds and prints all the division of a given number.

```

void main()
{
int number, i;
printf("enter the number to find divisors:");
scanf("%d",number);
printf("\nThe divisors of given number is:");
for(i=1;i<number;i++)
{
if(number%i==0)
    printf("%4d",i);
}
}

```

10. Write a C program to convert the given binary number into decimal number.

```

void main( )
{
long binary;
int dec=0, i=0;
printf ("\nEnter a binary No\n");
scanf ("%d", &binary);
while(binary>0)
{
dec +=(binary% 10)*pow(2,i++);
binary/=10;
}
printf (" Equivalent decimal number is %d", dec);
}

```

- 11. Using ASCII table, write a C Program that prints the following output using nested for loop.**

```
A  
A B  
A B C  
.....  
void main()  
{  
int i,j;  
for(i=0;i<26;i++)  
{  
    for{j=65;j<=65+i;j++)  
    { printf("%c", j); }  
    printf("\n");  
}  
}
```

- 12. Write a C program that prints out the even numbers 2,4,.....20 using for ..loop.**

```
void main()  
{ int i;  
clrscr();  
printf("Even number:");  
for( i=2;i<=20;i+=2)  
printf("%d",i); }
```

- 13. Write a program to print the triangle of the following format.**

```
1  
1 2  
1 2 3  
.....  
void main ( )  
{  
int i, n, j ;  
printf ("Type a number \n");  
scanf ("%d", &n);  
for (i=1 ; i<n ; i +){  
for (j=1 ; j<=i ; j +)  
{  
printf ("%3d", j);  
}  
printf ("\n");  
}
```

- 14. Write a C program to print all the Armstrong numbers between 1 and 500 , An Armstrong number is a number that is the sum of cubes of each of its digits is equal to the number itself.**

```
void main()
{
int i,num,sum,r;
printf("the Armstrong numbers between 1 and 500:\n\n");
for(i=1;i<=500;i++)
{
    sum=0;num=i;
    do
    {
        r=num%10;
        sum = sum+(r*r*r);
        num=num/10;
    } while (num!=0);
    if(sum==i)
        printf("%d",i);
}
}
```

- 15. Write a C Program to accept 10 integers from the keyboard and to display their sum and average using while loop.**

```
main()
[
int number[10],sum=0,i=0;
float average;
    while(i<10)
    {
        printf("enter the % d number:"i+1);
        scanf("%d",&number);
        sum+=number;
        i++
    }
average =sum/10.0;
printf("\n\n The sum of 10 integers:%d",sum);
printf("\nThe average of 10 integers:%2f", average);
}
```

- 16. Write a menu driven program to find perimeter of rectangle, circle, square, triangle**

```
void main( );
{
float l, b, r, side;
int x,y,z,c;
printf(" 1 for rectangle");
```

```

printf(" 2 for circle");
printf(" 3 for square");
printf(" 4 for triangle");
scanf("%d ", &c);
switch( c)
{
case 1:   scanf( "%f%f", &l &b);
            printf("%f ", l * b);
            break;
case 2:   scanf( "%f", &r);
            printf("%f ", 3.14 * r * r);
            break;
case 3:   scanf( "%f", &side);
            printf("%f ", 4 * side);
            break;
case 4:   scanf( "%f%f%f", &x &y &z);
            printf("%f ", (x + y + z)/2);
            break;
}
}

```

17. Write a program to count all prime numbers that lie between 100 to 200. [Note: A prime number is positive integer that is divisible only by 1 or by itself]

```

void main()
{
int i,j,count;
count=0;
clrscr();
printf("\n\nSeries of prime number from 100 to 200:\n");
for(i=100;i<=200;i++)
{
for(j=2;j<=i;j++)
{
if(i%j==0)
break;
}
if(i==j)
{
printf("%4d\n",i);
count+=1;
}}
printf("The countable number is: %d",count);
getch();
}

```

ARRAYS

18. Write an appropriate array definition for the following: define a two dimensional array (3*4)of integers called n. Assign the following values to the array elements.

10	12	14	16
20	22	24	26
30	32	34	36

```
int n[3][4]={  
    {10,12,14,16},  
    {20,22,24,26},  
    {30,32,34,36}  
};
```

19. Write a C Program to read a one-dimensional array “mark”of 10 elements. Print the above elements in reverse order.

```
#include<stdio.h>  
main()  
{  
    int mark[10];  
    clrscr();  
    printf("Enter the marks:\n");  
    for(i=0;i<5;i++)  
        scanf("%d",&mark[5]);  
    printf(" The Original array is:\n");  
    for(i=0;i<5;i++)  
        printf("%d      ",&mark[i]);  
    printf(" The reverse order is:\n");  
    for(i=4;i>=0;i--)  
        printfff("%d      ",mark[i]);  
    getch();  
}
```

20. Write a C program to find the sum of square of elements on the diagonal of a square matrix of order 5.

```
#include<stdio.h>  
#define MAX 5  
main()  
{  
    int mat[MAX][MAX],i,j,sum=0;  
    clrscr();  
    printf("enter the matrices elements:\n");  
    for(i=0;i<MAX;i++)  
        for(j=0;j<MAX;j++)
```

```

    {
        printf("enter the number:");
        scanf("%d",&mat[i][j]);
    }
    for(i=0;i<MAX;i++)
        sum+=mat[i][i];
    printf("The sum of diagonal element are:%d",sum);
    getch();
}

```

21. Write a program to find the average of 10 real numbers in an array.

```

#include<stdio.h>
#include<math.h>
void main()
{
    int i=0;
    float sum,avg,num;
    sum=0;
    while(i<10)
    {
        scanf("%f",&num);
        sum=sum+num;
        i++;
    }
    avg=sum/10;
    printf("Sum=%f\n",sum);
    printf("Average=%f\n",avg);
}

```

22. Write a C program to find if a number is present in a list of N numbers or not.

```

#include<stdio.h>
#include<conio.h>
void main()
{
    int i,n,m,flag=0; int a[10]; clrscr();
    printf("how many elements u want to enter");
    scanf("%d",&n);
    printf("enter element in the array");
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);
    printf("enter the element u want to search");
    scanf("%d",&m);
    for(i=0;i<n;i++)
    {

```

```

if(a[i]==m)
{
flag=1;
break;
}
if(flag==0)
printf("not present");
else
printf("present");
getch();}

```

23. Write a program to print bytes reserved for various types of data and space required for storing them in memory using array.

```

#include<stdio.h>
main()
{ int a[10]; char c[10];
float b[10];
printf("the type „int“ requires %d bytes", sizeof(int));
printf("\n The type „char“ requires %d bytes", sizeof(char));
printf("\n The type „float“ requires %d bytes", sizeof(float));
printf("\n %d memory locations are reserved for ten „int“ elements", sizeof(a));
printf ("\n %d memory locations are reserved for ten „char“ elements", sizeof(c));
printf (" \n %d memory locations are reserved for ten „float“ elements", sizeof(b)); }

```

Output:

The type „int“ requires 2 bytes
The type „char“ requires 1 bytes
The type „float“ requires 4 bytes
20 memory locations are reserved for ten „int“ elements
10 memory locations are reserved for ten „char“ elements
40 memory locations are reserved for ten „float“ elements.

24. Write a program using a single subscripted variable to read and display the array elements.

```

#include<stdio.h>
#include<conio.h>
main()
{ int i ;
float x[10];
printf("Enter 10 real numbers: \n");
for (i=0; i <10; i++)
{ scanf(" %f ", &x[i]); }
printf("The array elements are:");
for(i=0; i < 10; i++)
{ printf("%d \t", x[i]); }
getch();
}

```

25. Program to find out the largest and smallest element in an array.

```
# include<stdio.h>
#include<conio.h>
main( )
{ int i,n;
float a[50], large, small;
printf("size of vector/value:");
scanf("%d", &n);
printf(" \n vector elements are \n");
for(i=0; i<n; i++)
scanf(" %f ", &a[ i ]);
large = a[0];
small = a[0];
for(i=1; i<n; i++)
{ if(a[ i ] > large)
large = a[ i ];
else
if(a[ i ] < small)
small = a[ i ]; }
printf("\n Largest element in vector is %8.2f \n", large);
printf(" \n smallest element in vector is %8.2f \n", small);
getch( );
}
```

26. Program to sort the vector elements in ascending order.

```
# include<stdio.h>
# include<conio.h>
main( )
{ int i,j,k,n;
float a[50], temp;
printf("size of vector:");
scanf("%d", &n);
printf(" \n vector elements are : \n");
for( i=0; i < n; i++)
scanf(" %f ", &a[ i ]);
for( i=0; i < n-1; i++)
for( j=i+1;j<n; j++)
{
if(a[ i ] > a[ j ])
{
temp = a[ i ];
a[ i ] = a[ j ];
a[ j ] = temp;
}
}
printf("\n vector elements in ascending order : \n");
```

```

for( i=0; i<n; i++)
printf(" %8.2f ", a[ i ]);
getch();
}

```

27. Write a program to display the elements of two dimensional array.

```

#include<stdio.h>
#include<conio.h>
void main( )
{
int i,j;
int a[3][3] = { { 1,2,3}, {4,5,6}, {7,8,9}};
printf("elements of an array \n \n");
for( i=0; i<3; i++)
{
for ( j=0; j<3; j++)
printf ("%d\t", a[ i ][ j ]);
}
printf("\n"); /* end of outer for loop */
} /* end of main() function */

```

28. Program to read the matrix of the order upto 10 x 10 elements and display the same in matrix form.

```

#include<stdio.h>
#include<conio.h>
void main( )
{
int i, j, row, col, a[10][10];
printf("\n Enter Matrix Order upto (10 x 10) A :");
scanf("%d %d", &row, &col);
printf("\n Enter Elements of matrix A: \n");
for( i=0; i<row ; i++)
{
for( j=0; j<col; j++)
{ scanf("%d", &a[ i ][ j ]); }
}
printf(" \n The matrix is: \n");
for( i=0; i<row; i++)
{ for(j=0; j<col; j++)
{ printf(" %d ", a[ i ][ j ]); }
}
printf("\n"); }

```

29 . Program to perform addition & subtraction of two matrices, whose orders are upto 10 x 10.

```

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

```

```

main( )
{ int i,j,r1,c1, a[10][10], b[10][10];
clrscr( );
printf("Enter Order of Matrix A & B upto 10 x 10:");
scanf("%d %d", &r1, &c1);
printf("Enter Elements of Matrix of A: \n");
for( i=0; i < r1; i++)
{ for( j=0; j<c1; j++)
scanf(" %d ", &a[ i ][ j ]); }
printf("Enter Elements of Matrix of B: \ n");
for( i=0; i < r1; i++)
{ for( j=0; j < c1; j++)
scanf(" %d ", &b[ i ][ j ]); }
printf("\n Matrix Addition \n");
for( i=0; i < r1; i++)
{ for( j=0; j < c1; j++)
printf("%d\t", a[ i ][ j ] + b[ i ][ j ]); }
printf (" \n");
printf("n Matrix Subtraction/Difference \n");
for( i=0; i < r1; i++)
{ for( j=0; j < c1; j++)
printf("%d\t", a[ i ][ j ] - b[ i ][ j ]); }
printf("\n");
getch( ); }

```

30. Write a C program to multiply A matrix of order mxn with B matrix of order nxl.

```

# include<stdio.h>
# include<conio.h>
# include<math.h>
main( )
{ int a[10][10], b[10][10], c[10][10], m , n , i , j , l , k ;
clrscr( );
printf(" \n Enter Order of A matrix :");
scanf("%d %d", &m, &n);
/* loop to read values of A matrix */
printf("Enter a matrix \n");
for( i=0; i<m; i++)
for ( j=0; j<n; j++)
scanf("%d", &a[ i ][ j ]);
printf(" \n Enter order of B matrix:");
scanf(",%d %d", &n, &l);
/* loop to read values of B matrix */
printf("Enter B matrix \n");
for( i=0; i<n; i++)
for ( j=0; j<l; j++)
scanf("%d", &b[ i ][ j ]);
for( i=0; i<m; i++)

```

```

{ for ( j=0; j<l; j++)
{ c[ i ][ j ] = 0;
for( k=0; k < n; k++)
c[ i ][ j ] = c[ i ][ j ] +a[ i ][ k ] * b[ k ][ j ]; } }
printf(“\n Resultant matrix is \n”);
for( i=0; i<m; i++)
{ for( j=0; j<l; j++)
printf(“%6d”, c[ i ][ j ]);
printf(“\n ”); }
getch ( );
}

```

STRINGS

31. Write a program to find the length of a string without using string function

```

main()
{
    char str[50];
    int len=0,i=0;
    printf(“\n Enter the string.”);
    gets(str);
    while(str[i])
    {
        len++;
        i++;
    }
    printf(“\n The length of the given string is : %d”,len);
}

```

32. Write a ‘C’ program using conditional operator to print whether the character entered through the keyboard is a lower case letter or not.

```

#include<stdio.h>
#include<string.h>
main()
{
    int lower;
    char ch;
    printf(“Enter a character.”);
    ch=getchar();
    lower=ch>='a' && ch<='z' ? 1:0;
    if(lower)
        printf(“\n\nThe given letter is a lowercase letter.”);
    else
        printf(“\n\nThe given letter is not a lowercase letter.”);
}

```

33. Write a program to find the no of words in a given sentence.

```
#include<stdio.h>
main()
{
    char str[50];
    int word=1,i;
    clrscr();
    printf("\n Enter the sentence:");
    gets(str);
    for(i=0;i<strlen(str);i++)
        if((str[i]== ' '))
            ++word;
    printf("\n\n The no.of words in given sentence is: %d",word);
}
```

34. Write a ‘C’ program to arrange a list of names in alphabetical order.

```
#include<stdio.h>
#include<conio.h>
main()
{
    char Names[20][30],Temp[30];
    int i,n;
    printf("\n How many names?:")
    scanf("%d",&n);
    fflush(stdin);
    for(i=0;i<n;i++)
    {
        printf("\n Enter the %d person name:",i+1);
        gets(Names[i]); /*Get the names */
    }
    for(i=0;i<n-1;i++)
        for(j=i+1;j<n;j++)
    {
        if(strcmp(Names[i],Names[j])>0)
        {
            strcpy(Temp,Names[i]);
            strcpy(Names[i],Names[j]);
            strcpy(Names[j],Temp);
        }
    }
    for(i=0;i<n;i++)
    printf("\n The %d Person Name :%s",i+1,Names[i]);
}
```

- 35. Write a program to concatenate two strings without using string functions.**

```
#include<stdio.h>
main()
{
    char str1[50],str2[25];
    int i,j;
    clrscr();
    printf("Enter the first string:");
    gets(str1);
    printf("\n Enter the second string:");
    gets(str2);
    j=strlen(str1);
    for(i=0;i<strlen(str2);i++,j++)
        str1[j]=str2[i];
    str1[j]=NULL;
    printf("\n The concatenated string is: %s",str1);
    getch();
}
```

- 36. Write a ‘C’ program to accept 15 characters from the key-board. Check whether they are valid lowercase alphabets and if so, convert them and display them as uppercase alphabets. If entry is other than lowercase, the character entered should be skipped and not displayed.**

```
#include<stdio.h>

main()
{
    char str[15];
    int i;
    printf("Enter the string");
    gets(str);
    printf("\n\nThe output string is:");
    for(i=0;i<15;i++)
        if((str[i]>='a') && (str[i]<='z'))
            printf("%c",str[i]-32);
}
```

- 37. Write a program to accept a string of six characters from the keyboard**

- Construct a new string with a blank space in between each character.**
- Output the original string in the reverse order.**

```
#include<stdio.h>
#include<conio.h>
main()
{
    char str[10],newstr[15];
```

```

int i,j=0;
clrscr();
printf("\n Enter the string:");
for(i=0;i<6;i++)
    str[i]=getchar();
str[i]=NULL;
for(i=0;i<6;i++,j+=2)
{
    newstr[j]=str[i];
    newstr[j+1]('');
}
newstr[j]=NULL;
printf("\n\n The new string is : %s",newstr);
printf("\n\n The reverse of original string is,");
for(i=strlen(str)-1;i>=0;i--)
    printf("%c",str[i]);

```

38. Given the string “DATA PROCESSING”, write a C program to read the string from terminal and display the same in the following formats.

(i) DATA PROCESSING

**(ii) DATA
PROCESSING**

```

#include<stdio.h>
#include<conio.h>
void main()
{
    int i;
    char a[50];
    clrscr();
    printf("enter the string");
    gets(a);
    puts(a);
    i=0;
    while(a[i]!=' ')
    {
        printf("%c",a[i]);
        i++;
    }
    i++;
    printf("\n");
    while(a[i]!='\0')
    {
        printf("%c",a[i]);
        i++;
    }
    getch();
}

```

- 39. Write a C program that reads a fixed point real number in a character array and then displays rightmost digit of the integral part of number.**

```
void main()
{
char a[10];
int i;
printf("enter a fixed point real no. : ");
scanf("%s",a);
i=0;
while(a[i]!='.')
i++;
printf("%c",a[--i]);
}
```

- 40. Write a program to display character array with their address.**

```
void main( )
{ char name[10] = {"A", "R", "R", "A", "Y"};
int i = 0;
printf("\n character memory location \n");
while(name[i] != "\0")
{ printf("\n [%c] \t [%u]", name[i], &name[i]); i++; }
```

Output:

Character Memory Location

[A]	4054
[R]	4055
[R]	4056
[A]	4057
[Y]	4058

- 41. Write a C program to count the occurrence of a particular character in the given string.**

```
void main( )
{
char st[20], ch; int count, l,i;
printf("\n Enter the string:");
gets(st);
printf("\n which char to be counted ?");
scanf(" %c", &ch); /
l = strlen(st);
count = 0;
for( i=0; i < l; i++)
if(st[ i ] == ch) count++;
printf("\n the character %c occurs %d times", ch, count);
}
```

SUMMARY

- Control statements are used to specify the order in which the various instructions in a program are to be executed.
- Control statements are classified into (i) Conditional statements (ii) Looping statements (iii) Unconditional statements
- *Examples for Conditional statements* are: If statement If – else statements , Nested if – else statements , Switch statement
- Examples for looping statements (i) While loop (ii) Do – while loop (iii) For loop.
- Examples for Unconditional statements (i) Break statements (ii) Continue statements (iii) goto statements
- The simple if statement will execute a single statement, or a group of statements, if the condition is true. If the condition is false, it does nothing.
- If ...else statement is used to execute one group of statements if the condition is true. The other group of statements are executed when the condition is false.
- A nested if statement is an if statement which is within another if – block or else – block.
- Looping statements are used to execute a group of instructions repeatedly until some condition is satisfied.
- The minimum number of times the body of a do... while loop is executed is one.
- A while loop performs its test before the body of the loop is executed, whereas a do..loop makes the test after the body is executed.
- The switch statement is used to pickup or executes a particular group of statements. It allows us to make a decision from the number of choices.
- The four keywords associate with switch statement are: (i) switch (ii) case (iii) default and (iv) break.
- The break terminates the loop. The continue branches immediately to the test portion of the loop.
- A loop with no terminating condition is called an infinite loop.
- The type of loops, where the number of the execution is known in advance are termed by the counter controlled loop.
- The type of loop where the number of execution of the loop is unknown, is termed by sentinel controlled loop.
- The goto statement is used to transfer the statements execution from one place to another.
- A switch statement is generally best to use when you have more than two conditional expressions based on a single variable of numeric type.
- A statement is a single C expression terminated with a semicolon. A block is a series of statements, the group of which is enclosed in curly-braces.
- The break statement unconditionally ends the execution of the smallest enclosing while, do, for or switch statement.
- Array is an ordered collection of elements that share the same name.
- The elements of the array are known as members of the array
- The dimension of an array is determined by the number of subscripts needed to identify each element.

- The one-dimensional array is also called as list.
- The subscript of the first element of an array in ‘C’ is zero.
- One dimensional array has a single subscript. But two dimensional arrays have two subscripts. The two dimensional array is called a matrix or table.
- Arrays can be classified into: (i) One-Dimensional arrays (ii) Two-Dimensional arrays and (iii) Multi-Dimensional arrays.
- An array of three or more dimension is called as multidimensional array.
- Any group of characters enclosed in double quotes is a string constant. It is terminated by a null character (\0)
- In ‘C’ language a string is defined by using the character array.
- When declaring character arrays, always allow one extra element space for the null character.
- There are two ways to read a string from the terminal. They are (1) by using the scanf function and (2) by using gets function.
- The disadvantage of scanf function is that there is no way to enter a multi-word string into a single variable.
- *A group of characters defined between double quotation marks is called as a string constant*
- Whenever a string is stored in memory, the NULL character ('\0') is automatically added to its end
- The most commonly used string-handling functions are strcat(), strlen(), strcmp, strcpy().
- The strcat() function is used to join two strings.
- strcmp() function compares two strings character by character(ASCII comparisons) and returns a value. Value may be less than 0, or equal to 0, or greater than 0.
- strcpy () copies the second string to the first string specified in the strcpy () parameter.
- strlen() function is used to find the number of characters in string.

REVIEW QUESTIONS / PROGRAMS

PART – A (2 Marks)

1. What do you mean by control statements in C?
2. How the control statements are classified?
3. What is the purpose of the comma operator? Within which control statement does the comma operator usually appear?
4. Write down the syntax of if-else statement.
5. What is meant by looping? Give an example.
6. Write an infinite loop in C language , which does not use any variable or constant/
7. What is nested loops ? Give an example.
8. State the difference between Counter controlled loop and sentinel controlled loop?
9. In a control structure switch-case, explain the purpose of using default.
10. What are the four keywords used in switch structure?
11. In what way does an array differ from an ordinary variable?
12. How are individual array elements identified?
13. What is an array? Can array index be negative?
14. What condition must be satisfied by the entire elements of any given array?
15. How a string is stored in C?
16. State the difference between character constant and string constant.
17. What is null character? State the use of null character.
18. What is the difference between a, "a" and ""a""
19. How do we read string including blank space?
20. State any four string manipulating functions.
21. Give the general structure of (i) strlen() (ii)strcmp() functions

PART – B (3 Marks)

1. Explain the forward and backward jump with necessary example.
2. What is the purpose of switch statement? What labels, i.e., case prefixes? What type of expression must be used to represent a case label?
3. What is time delay loop ? How you will form time delay loop?
4. In any program, using switch statement, if all break statements are removed from all cases of switch statement, how does it affect the functionality of switch statement? Give example.
5. What are subscripts? How are they written? What restrictions apply to the values that can be assigned to subscripts?
6. What is the difference between scanf() with % s and gets()?
7. What is the difference between character array and string?
8. Write a C Program that will capitalize all the letters of a string.
9. Write a C program to read a word and rewrite it in reverse order.
10. Write a program to find the largest number between given three numbers

PART – C (5 Marks / 10 Marks)

1. Describe the different forms of the if statement. How do they differ? Give examples.
1. Explain unconditional control statements with examples.
2. Draw the flow chart and syntax of else-if ladder.
3. Distinguish between Break and continue statement.
4. Write down the differences between do...while and while loop? Explain with an example.
5. Explain the general syntax of a switch statement and with simple example. Explain its use in programming.
6. What is the purpose of while-loop , do-while loop and for loop ? How these are executed? Summarize the rules associated with these structures.
7. What is the purpose of “continue” statement? Within which control structures can the “continue” statement be included? Compare it with the “break ” statement.
8. Explain (i) Entry controlled looping statement and (ii) exit controlled statement?
9. What are the different form of for loop ? Explain.
10. Write a program to find the factorial of a given number using the for loop.
11. Read a list of 10 numbers and print it in reverse order.
12. Read a 3 x 3 matrix and find their sum.
13. How can a list of strings be stored within a two dimensional array? How can individual strings be processed? What library functions are available to simplify the string processing?
14. Write a C program to print the quotient of an integer number without using ‘/’.
15. Write a C program to print all the even and odd numbers of a certain range as indicated by the user.
16. Illustrate the initialization of one dimensional arrays , two dimensional arrays and strings.
17. Take input from the user in a two-dimensional array and print the row-wise and column-wise sum of numbers stored in a two-dimensional array.
18. Swap the kth and (k+1) th elements in an integer array, k is given by the user.
19. What are the functions used for reading and writing strings? Explain them.
20. Write in detail about the following string handling functions: (i) strcat() (ii) strcpy() (iii) strlen() (iv)strcmp
21. Write a program to count number of characters, words and lines in a given text.
22. Write a program to count number of vowels, consonants and blanks in a given text.
23. Write a program to check whether a give string is palindrome or not.
24. Write a C program that takes an organization name as input and print it in abbreviation from. For example if the input is “ Bharath Heavy Electricals Limited ”, the the output should be BJEL.
25. Write a C program to find the Fibonacci series upto 200.
26. Write a C program to arrange the given N numbers in ascending order.
27. Write a C program to accept a month number and display the month name.
28. Write a program to delete an element from an array.

UNIT - III

FUNCTIONS , STRUCTURES AND UNIONS

OBJECTIVES

After reading this unit, the student will be able to

- Understand what a function is and how its use benefits a program
- Classify functions
- Learn how a function declaration, function call and function definition are constructed.
- Understand how arrays and functions are passed to function
- Classify different types of user defined functions.
- Understand what scope rules mean in functions,
- Learn about scope and lifetime of different types of variables.
- Understand the basic concept of recursion.
- Learn the techniques of constructing recursive function.
- Understand the basic concept of structures.
- Access, Initialize and copy structures and their members
- Understand nesting of structures
- Learn about union data type.
- Differentiate structures from union

INTRODUCTION

A function is a self-contained block of program statements that perform a particular task. Every C program is a collection of functions. A large single list of instructions becomes difficult to understand, debug, test and maintain. For this reason, functions are used. The function code is stored in only one place in memory, even though it may be executed as many times as a user needs thus saving both time and space.

A structure is a derived data type. Structure is a collection of mixed data types referred by a single name. Closely associated with the structure is the Union, which also contains multiple members. Unlike a structure, the members of a union share the same storage area.

In this unit, we will discuss about the concepts of functions, structures and unions.

3.0. FUNCTIONS - DEFINITION

A function in C is a **small “sub-program”** that performs a particular task. C, functions can be classified into two categories, namely, (i) Library functions or Built in functions or Pre-defined functions and (ii) User-defined functions.

Pre-defined functions are already written by compiler developers. Pre-defined functions are not written by the programmer. Pre-defined functions are commonly used in all the programs. These functions are grouped together and stored in a library.

Programmer defined functions can be written by the programmer at the time of writing a program.

3.1. BUILT -IN - FUNCTIONS

C contains number of library functions. These functions are also known as built-in functions. The related library functions are grouped together and stored in a header file. Some of the library functions are shown below

Header file	Library function	Header file	Library function
stdio.h	gets, puts, getchar, putchar, scanf, printf	ctype.h	tolower, toupper, isalpha, isdigit
string.h	strcat, strlen, strcmp, strcpy	Stdio.h	printf(), scanf(), getc, gets, putc, putc
math.h	sin, cos, pow, ceil, floor	math.h	

3.1.1. Math Functions

sin(sine) : This function is used to find the sine value. The value for sine must be in radians. This function returns a value in the range -1 to 1.

Syntax: double sine(double x);

cos(cosine) : This function is used to find the cosine value. The value for cosine must be in radians. This function returns a value in the range -1 to 1.

Syntax: double cos(double x);

tan(tangent) : This function is used to find the tangent of a value. The value for tangent must be in radians. This function returns a value in the range -1 to 1.

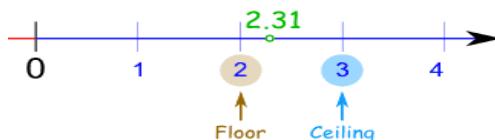
Syntax: double tan(double x);

ceil (roundsup): Ceil function returns the smallest integer greater than or equal to the given value

Syntax: double ceil(double x);

floor (roundsdown) : floor function returns the largest integer less than or equal to the given value.

Syntax: double floor(double x);



Some example:

x	Floor	Ceiling
-1.1	-2	-1
0	0	0
1.01	1	2
2.9	2	3
3	3	3

exp(Exponent) : Exp function calculates 'e' to power of xth power.

Syntax: double exp(double x);

abs(absolute): This function is used to find the absolute value of an integer.

Syntax: int abs(int x);

pow(power) : This function is used to find the power of any value(x) to any value (y). (i.e. x^y)

Syntax: double pow(double x,double y);

Example : pow(5,2) = 25

sqrt(square root) : This function is used to find the square root of any number.

Syntax: double sqrt(double x);

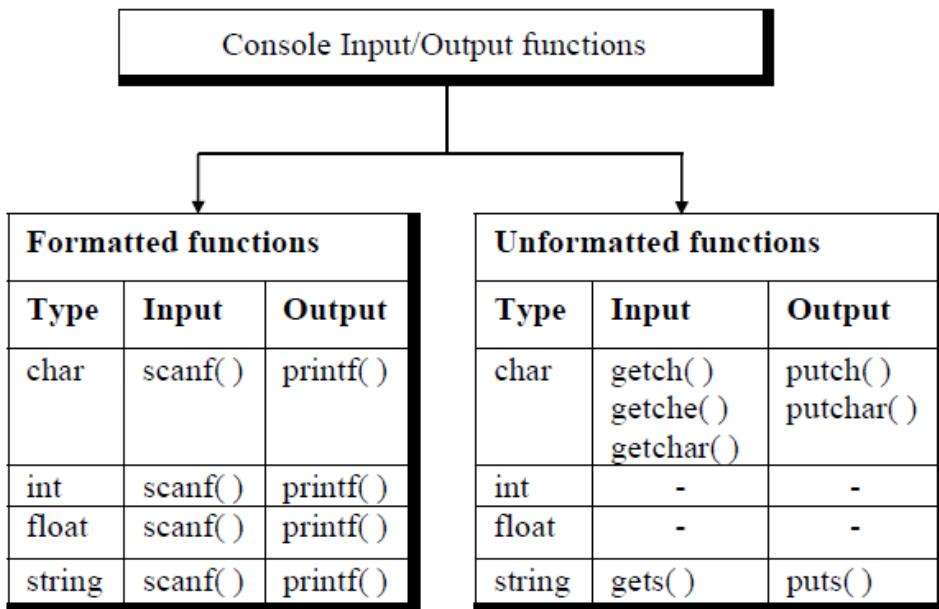
Examples by using functions in math.h header file

```
void main ()  
{ double Degree, Radian;  
double w,x,y,z;  
double pi = 22.0 / 7.0;  
printf ("\nEnter the angle in degrees:");  
scanf ("%f", &Degree);  
Radian = (pi / 180.0* Degree);  
x = sin(Radian);  
y = cos(Radian);  
z = tan(Radian);  
w = exp(x);  
printf ("\nSin(%f)=%f\nCos(%f)=%f\n Tan (%f)=%f\nExp(%f)=%f",  
Degree, x, Degree, y, Degree, z,x,w);  
}
```

Output

```
Sin(48.250000)= 0.746283  
Cos((48.250000f)=0.665629  
Tan ((48.250000)=1.121170  
Exp(0.746283)=2.109146
```

3.1.2. Console functions



The screen and keyboard together are called a console. Console I/O functions can be further classified into two categories—formatted and unformatted console I/O functions. The basic difference between them is that the formatted functions allow the input read from the keyboard or the output displayed on the VDU to be formatted as per our requirements. For example, if values of average marks and percentage marks are to be displayed on the screen, then the details like where this output would appear on the screen, how many spaces would be present between the two values, the number of places after the decimal points, etc. can be controlled using formatted functions.

Unformatted console I/O functions work faster since they do not have the overheads of formatting the input or output.

3.1.3. Character functions

The library functions contained in the header file ctype.h are called as Character Class Test Functions. These functions are used to find the type of the character and used for character conversion. Some of the functions available in the above header file are described below:

isdigit(x) : This function is used to check whether a given character is a digit or not. This function returns a non-zero integer for true and zero for false.

Syntax :	int isdigit(int x);
----------	---------------------

isalpha(x) : This function is used to check whether a given character is a letter or not. This function returns a non-zero integer for true and zero for false.

Syntax :	int isalpha(int x);
----------	---------------------

islower(x) : This function is used to check whether a given character is a lowercase or not. This function returns a non-zero integer for true and zero for false.

Syntax :	int islower(int x);
-----------------	---------------------

isupper(x): This function is used to check whether a given character is a uppercase or not. This function returns a non-zero integer for true and zero for false.

Syntax:	int isupper(int x);
----------------	---------------------

ispunct(x) : This function is used to check whether a given character is a punctuation character or not. This function returns a non-zero integer for true and zero for false.

Syntax :	int ispunct(int x);
-----------------	---------------------

tolower() : Converts character to lowercase

toupper() : Converts character to uppercase

islower() and tolower(): islower(c) determines if the passed argument is lowercase. It returns a nonzero value if true otherwise 0. tolower () is a conversion function , that convert argument c to lowercase

Example

```
main()
{
    char v = 'A';
    char w = '6';
    char x = 'q';
    char y = 'Q'
    clrscr();
    if(isalpha(v)) printf("\n%c is an alphabet",v);
    if(isdigit(w)) printf("\n%c is a digit",w);
    if(islower(x)) printf("\n%c is a lowercase character",x);
    if(isupper(y)) printf("\n%c is a uppercase character",y);}
```

OUTPUT

A is an alphabet

6 is a digit

q is a lowercase character

Q is a uppercase character

3.2. USER DEFINED FUNCTIONS

3.2.1. Introduction about functions

A function is a self-contained program segment that performs a particular task. A large program has to be split into smaller segments so that it can be efficiently solved. Functions are smaller segments, which are used to solve the large problem.

3.2.2. Reasons for using the functions

1. *A function avoids the need for writing repeated codes having same statements.* The length of a source program can be reduced by using functions at appropriate places.
2. *Functions are easier to write, debug and understand.* Simple function can be written to perform unique specific task. Programs containing the functions are also easier to maintain. Functions are also put in a library and later used by many programs.
3. *Understanding the flow of program and its code is easy,* since the readability is enhanced while using the functions.
4. A single function written in a program can also be used by other programs.
5. Make program self-documenting and readable.

3.2.3. Elements of user defined functions

The following are the three elements of user defined function

- 1. Function definition**
- 2. Function call**
- 3. Function declaration**

The function definition is an independent program module that is specially written to implement to the requirements of the function. Function call is used to invoke it at a required place in the program. The program or a function that has called a function is referred to as the calling function or calling program. The calling program should declare any function that is to be used later in the program. This is known as the function declaration

3.2.4. Function definition

A function definition, also known as function implementation shall include the following elements

- 1. Function name**
- 2. Data type**
- 3. List of parameters (Argument list(s))**
- 4. Local variable declarations**
- 5. Function statements**
- 6. A return statement**

All the six elements are grouped into two parts; namely, (i) Function Header (First three elements) and (ii) Function Body (Second three elements)

The function definition is the actual body of the function. The general form of a function is

```
[data type] function name (argument list)
{
    local variable declarations;
    statements;
    [return expression]
}
```

Function Header :

The function header consists of three parts; function type, function name and list of parameter.

(a) Data Type : The function type specifies the type of value (like float or double) that the function is expected to return to the calling program. If the return type is not explicitly specified, C will assume that it is an integer type.

(b) Function name : The function name is any valid C identifier and therefore must follow the same rules of formation as other variable names in C. The name should be appropriate to the task performed by the function.

(c) List of Parameter : The parameter list declares the variables that will receive the data sent by the calling program. They serve as input data to the function to carry out the specified task.

Function body :

The function body is enclosed in braces, contains three parts, in the order given below:

1. **Local variable declaration** : Local variable declarations are statements that specify the variables needed by the function.
 2. **Function Statements** : Function statements are statements that perform the task of the function.
 3. **Return Statements** : A return statement is a statement that returns the value evaluated by the function to the calling program. If a function does not return any value, one can omit the return statement.

Arguments appearing in the parentheses of function header are called as formal parameters or formal arguments or dummy arguments. The body of the function may consist of many statements.

Example

```
int sum ( int a, int b)
{
    int c;
    c = a+b;
    return c;
}
```

The above is the function definition for the function “sum”. The function takes up two integers namely “a” and “b”. They are called as dummy parameters. The function returns an integer value. Some functions will not actually return a value or need any arguments. For these functions the keyword **void** is used.

3.2.5. Return Statement

Information is returned from the function to the calling program through a return statement. Return statement has two purposes.

1. Executing of return statement immediately transfers control from the function to the calling program
2. The value inside the parentheses of return statement is returned to the calling program.

The general formats of return statements are

- return;
- return (constant);
- return (expression);
- return (variable);
- return (condition expression);

The first statement does not return any value; it is just equal to the closing brace of the function.

Examples :

```
return(5);      return(x*y);    return (p);     return(a>b?a:b)
```

1. A limitation of return statement is that it can return only one value
2. The return statement need not always be present at the end of the called function.
3. If the called function should not return any value, then the keyword **void** must be used as data-type specifier.
4. It is possible for a function to have **multiple return statements**. For example:

```
Int large big(int a, int b)
{
    if (a>b)
        return a;
    else
        return b;
}
```

3.2.6. Function Declaration (Function prototyping)

The program or a function that called a function is referred to as the calling function or calling program. The calling program should declare any function that is to be used later in the program. This is known as the function declaration. Whenever a function is called, before to that the called function should be either declared or defined. Function declaration is the activity that is telling the compiler that a function is defined somewhere below.

A function declaration consists of four parts. They are,

1. Function type
2. Function name
3. Parameter list
4. Terminating semicolon

Function declaration can be written as

```
Function_type function_name(parameter list);
```

where function type refers to the data-type of the data returned by the function, and name refers to the function name. The declaration of a function is known as function prototype

If the function definition precedes all function calls, then it is not necessary to include a function declaration within the calling portion of the program.

Example

```
main( )
{ float square ( float ) ; /* function declaration stement */
    float a, b ;
    printf ( "\nEnter any number " ) ;
    scanf ( "%f", &a ) ;
    b = square ( a ) ;
    printf ( "\nSquare of %f is %f", a, b ) ; }

float square ( float x ) {
    float y ;
    y = x * x ;
    return ( y ) ;
}
```

Output:

```
Enter any number 1.5
Square of 1.5 is 2.250000
Enter any number 2.5
Square of 2.5 is 6.250000
```

Some points to keep in mind when calling functions

1. The number of arguments in the function call must match the number of arguments in the function definition.
2. The type of the arguments in the function call must match the type of the arguments in the function definition.
3. The actual arguments in the function call are matched up in-order with the dummy arguments in the function definition.
4. The actual arguments are passed by-value to the function. The dummy arguments in the function are initialized with the present values of the actual arguments.
5. *Any changes made to the dummy argument in the function will NOT affect the actual argument in the main program.*

3.2.7. Calling a function

A function can be called by specifying its name, followed by a list of arguments enclosed within parentheses and separated by commas. The parenthesis is used to indicate the compiler that the identifier is a function and not a variable. A function name used in the calling program is either a part of a statement or a complete statement with a semi-colon. **In function definition, semi-colon is not placed at the end of header.** The absence of semicolon indicates that the function is being defined and not called. If the function call does not require any arguments, an empty pair of parentheses must follow the function's name.

Important points to be considered when calling a function

1. A semicolon is used at the end of the statement when a function is called.
2. Parentheses are compulsory after the function name.

3.2.8. Function Call

Whenever a function uses the service of another function, then the former one is called as “calling function” and the later one is called as “called function”. The process of invoking a function is referred as “Calling a function”. During the function call, the execution of current function is temporarily stopped and the control moves to the execution of the function called. While calling the function, some values are provided within (). These values are called as actual parameters or actual arguments.

The above values are copied or assigned to the dummy parameters available in the function header in order. The number of actual parameters should be equal to the number of dummy parameters and their type should also be same.

After the control is moved to the called function, the execution of the called function starts. The statements in the function are executed upto return statement. Finally the result is returned to the calling function.

Example

```
main()
{
    int l;
    for (i=1; i<=10; i++)
    {
        j = square(i); /* function call */
        printf("\nSquare of %d is %d", l,j);
    }
}
square ( int x)
{
    int y;
    y = x*x;
    return(y);
}
```

The above program is used to print the squares of 1 to 10. This is achieved by calling a function called square. The result is passed from the called function.

3.2.9. Types of functions

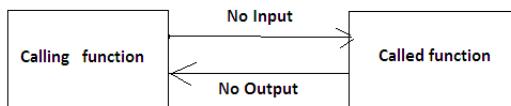
A function may belong to any one of the following categories:

1. Functions with no arguments and no return values.
2. Functions with no arguments and return values
3. Functions with arguments and no return values.
4. Functions with arguments and return values.

1. Functions with no arguments and no return values:

There is no data transfer between the calling function and the called function. When a function has no arguments, it does not receive any data from the calling function. Similarly, when it does not return value, the calling function does not receive any data from the called function.

A function that does not return any value cannot be used in an expression. It can be used only as independent statement.

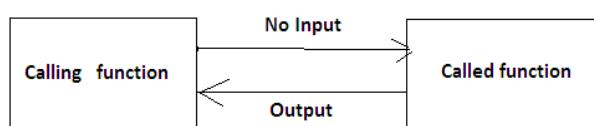


Example:

```
/* Program to illustrate a function with no argument and no return values*/
#include <stdio.h>
main()
{
statement1();
starline();
statement2();
starline();
}
statement1()
{
printf("\n Sample subprogram output");
}
statement2()
{
printf("\n Sample subprogram output two");
}
starline()
{
int a;
for (a=1;a<60;a++)
printf("%c",'*');
printf("\n");
}
```

2. Functions with No arguments but return values:

When a function has no arguments, it does not receive any data from the calling function. When it returns a value, the calling function receives the data from the called function.

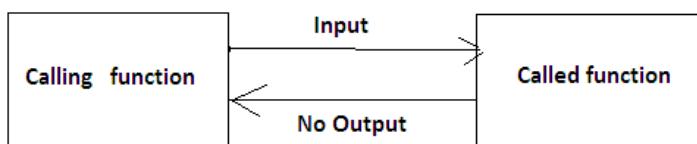


Example:

```
#include <stdio.h>
float area()
{
int radius;
float area;
printf("Enter Radius of the circle\n");
scanf("%d",&r)
area = 3.14 *r * r;
return(area);
}
void main()
float ar;
ar = area();
printf(" Area of the circle is %f", ar);
}
```

3. Functions with arguments but no return values:

When a function has arguments, it receives data from the calling function. The calling function can read data from the input terminal and pass it to the called function.



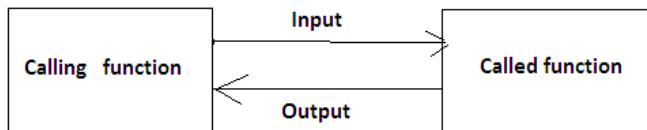
```
/*Program to find the largest of two numbers using function*/
#include <stdio.h>
main()
{
int a,b;
printf("Enter the two numbers");
scanf("%d%d",&a,&b);
largest(a,b) ;
}
/*Function to find the largest of two numbers*/
largest(int a, int b)
{
if(a>b)
printf("Largest element=%d",a);
else
printf("Largest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

In the above program we could make the calling function to read the data from the terminal and pass it on to the called function. But function does not return any value.

4. Functions with arguments and return values:

The function of the type Arguments with return values will send arguments from the calling function to the called function and expects the result to be returned back from the called function back to the calling function.

Example:



```
main()
{
float x,y,add();
double sub();
x=12.345;
y=9.82;
printf("%f\n" add(x,y));
printf("%lf\n"sub(x,y);
}
float add(a,b)
float a,b;
{
return(a+b);
}
double sub(p,q)
double p,q;
{ return(p-q); }
```

Call by value

When the values are passed to a function via an actual argument, the value of the actual argument is copied into the function. Therefore, the values of the corresponding formal arguments can be altered within the function. But the values of the actual argument within the calling function will not change. The procedure for passing the values of arguments to a function is known call by value.

Advantages of call by value

1. Expression can be passed as arguments.
2. Unwanted changes to variables in calling program can be avoided.

Disadvantages of call by value

1. Information cannot be passed back to calling function through arguments.

Example

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

```

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

```

The output of the above program would be:

```
x = 20 y = 10
a = 10 b = 20
```

Note that values of **a** and **b** remain unchanged even after exchanging the values of **x** and **y**.

3.2.11. The Scope and Life time of variables used in functions

STORAGE CLASSES

Every variable in C has two attributes: its data type and its storage class. The storage class refers to the **way memory is allocated** for the variable. The storage class also determines the **scope of the variable**, that is, what parts of a program the variable's name has meaning. In C, the four possible Storage classes are

- auto
- extern
- static
- register

The life time of a variable refers to the existence of a variable in memory. The four classes of a variable, as classified by their scope and life time are

1. Automatic Variables
2. External variable
3. Static variable
4. Register variable.

Automatic variables

Automatic variables are declared inside a function. They are created as soon as a function starts execution, and used within the function. At the end of the execution of the function, it is destroyed. Here “created” means the memory is allotted to the variable and “destroyed” means the memory is freed and released back to operating system.

Automatic variables are therefore local to the function and are not recognized outside the function. So, automatic variables are also called as “local variable” or “internal variable”.

A keyword “auto” is sometimes used to specify automatic variable storage class. Automatic variables are initialized by the programmer. They are recreated each time the function is called.

Example

```
/* A program to illustrate the working of auto variables*/
#include
void main()
{
int m=1000;
function2();
printf("%d\n",m);
}
function1()
{
int m=10;
printf("%d\n",m);
}
function2()
{
int m=100;
function1();
printf("%d\n",m);
}
```

Output

```
10
100
1000
```

The output clearly shows that value assigned to m in one function does not affect its value in the other function. The local value of m is destroyed when it leaves a function.

External variables

External variables are active throughout the program execution. They are also known as *global variables*. External variables can be accessed by any function in the program. External variables are declared outside a function. If a variable is declared at the beginning of a program outside all functions [including **main()**] it is classified as an external by default.

External variables are useful when variables have to be shared among functions.

If a variable is declared as external variable, any function can use it and change its value. Subsequent functions can reference only that new value.

Example

The following example illustrates the use of global variable.

```
int j ;
main( )
{
    printf ( "\nj = %d", j ) ;
    increment( );
    increment( );
```

```

        decrement( );
        decrement( );
    }
increment( )
{
    j = j + 1 ;
    printf ( "\incrementing j = %d", i ) ;
}
decrement( )
{
    j = j - 1 ;
    printf ( "\decrementing j = %d", i ) ;
}

```

Output :

```

j = 0
incrementing j = 1
incrementing j = 2
decrementing j = 1
decrementing j = 0

```

The value of j is available to the functions **increment()** and **decrement()** since j has been declared outside all functions.

Advantage of using global variables: It is a method of transmitting information between functions in a program without using arguments.

Static variables

The value of the static variable persists until the end of the program. When variables are required to retain their values from one execution to another execution of the function, they should be declared with the keyword “static” before their type declaration.

Static variable will not lose their storage locations or their values when control leaves the functions or blocks wherein they are defined. The initial value assigned to a static must be a constant or an expression involving constants.

Static variables are automatically initialized to 0, by default during the first execution of the program. However, the initialization does not make place during subsequent calls.

Example

```

main()
{
    int j;
    for(j=1;j<3;j++)
        inc();
    }
    inc();
    {
        static int x=0;
        x=x+1;
    }
}

```

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

Output : 1 2 3

During the first call to inc() in the example shown above , x is incremented to 1. because x is static, this value persists and therefore the next call adds another 1 to x giving it a value of 2. The value of x becomes 3 when third call is made. If we had declared x as an auto then output would here been x=1 all the three times.

REGISTER VARIABLES

Registers variables are placed in one of the machine registers, instead of using memory. Accessing the register is very fast compared to memory. So frequently used variables are placed in register. This will increase the execution speed. The general format to declare a variable as register variable is

register data_type variable_name

These variables are defined with the keyword “register”. Register variables should be having maximum size of the computer register. Suppose if the register size is 4 byte means , it is not possible to go for variables whose size is greater than 4 bytes. Thus char, int, long and float are possible types. *Double and long double are not possible for this storage type.*

Example

```
main()
{
    register int count;
    ----
}
```

Register variables are used to speed up the program execution. It is error to refer the address of a register variable.

COMPARISON OF STORAGE CLASSES IN C

a. **Automatic storage class:** The features of variables are as follows

- Storage: Memory
- Default initial value: Garbage value
- Scope: Local to the block in which defined
- Life: till the control remains within the block in which defined.

b. **Register storage class:** The features of variables are as follows

- Storage: CPU registers
- Default initial value: Garbage value
- Scope: Local to the block in which defined
- Life: till the control remains within the block in which defined

c. **Static storage class:** The features of variables are as follows

- Storage: Memory
- Default initial value: Zero
- Scope: Local to the block in which defined
- Life: value of variable persists between different function calls.

d. **External storage class:** The features of variables here are as follows

- Storage: Memory
- Default initial value: Zero
- Scope: global
- Life: As long as program execution does not come to an end.

3.2.12. Recursion

Recursion is the process in which a function repeatedly calls itself to perform calculations. For example consider the following:

```
main()
{
    printf("This is an example of recursion.\n");
    main(); }
```

When executed this program will produce an output which is something like this,

This is an example of recursion.

This is an example of recursion.

Execution is terminated abruptly; otherwise the execution will continue indefinitely.

Recursion is a special case of function call where a function calls itself. These are very useful in the situations where solution can be expressed in terms of successively applying same operation to the subsets of the problem. For example, a recursive function to calculate factorial of a number n is given below:

The following function calculates factorials recursively:

```
int fact(int n)
{
    int factorial;
    if(n==1 || n==0)
        return(1);
    else
        factorial=n*fact(n-1);
    return (factorial);
}
```

Assume n=4, we call fact(4)

Since n is not equal to 1 0, factorial=n*fact(n-1)

Factorial=4*fact(3) (again call fact function with n=3)

=4*3*fact(2) (again call fact function with n=2)

=4*3*2*fact(1) (again call fact function with n=1)

=4*3*2*1 (terminating condition)

=24

Always have a terminating condition with a recursive function call otherwise function will never return.

3.3.STRUCTURES AND UNIONS

3.3.1. Structures - Introduction

Structure is a compound data type. It is used to store different types of data items. *Sometimes a structure is called as a record.* The individual structure element is referred to as “members”. A structure is a convenient method of handling a group of related data items of different data types.

Differences between Array and structure

Array	Structure
Array is a collection of homogeneous data.	Structure is a collection of heterogeneous data.
Array elements are referred by subscript.	Structure elements are referred by its unique name.
Array elements are accessed by it's position or subscript.	Structure elements are accessed by its object as '.' operator.
Array is a derived data type.	Structure is user defined data type.

3.3.2. Defining a structure

The general format to define a structure is as follows.

```
struct tag_name
{
    data_type member1;
    data_type member2;
    ...
    ...
};
```

Here ‘struct’ is the keyword to declare a structure. tag_ name can be any user-defined identifier. The structure declaration should always end with ‘;’.

Example

Consider a library database consisting of title of the book, autor name, number of pages and price. A structure used to define the above is

```
struct lib_books
{
    char title[20];
    char author[15];
    int pages;
    float price;
};
```

The above example declares a structure “lib_books” with 4 members : title, author, pages and price.

3.3.3. Variable Declaration

The compiler does not reserve any memory space when a structure is declared. Memory space is reserved when only a variable of this type is defined.

Structure variable is defined anywhere in the program by using the tag-name. For example, the statement

```
struct lib_books c, java, basic;
```

defines c, java, basic as variables of type struct lib_books..

The syntax for creating the structure variable is

```
struct tag_name variable_name1, variable_name2,.....
```

The memory allocated for a structure variable is equal to the sum of memory allocated for all its members.

For the above example 41 byte are allocated, for each variable, which is calculated as follows.

title 20 bytes (20 * 1) author- 15 bytes pages - 2 bytes price - 4 bytes

It is also possible to combine both definition and variable declaration in the same statement. For example

```
struct lib_books
{
    char title[20];
    char author[15];
    int pages;
    float price;
} c, java, basic;
```

is also valid. The use of tag-name is optional when declaring the variables during structure definition.

3.3.4. Initialization of a structure

The members of a structure can be initialized to constant values by enclosing the values to be assigned within the braces after the structure definition. Thus the declaration

```
struct date
{
    int date;
    int month;
    int year;
} republic = {26,1,1950};
```

initializes the member variables date, month, year of republic to 26,1,1950 respectively.

Struct date Gandhi_jay = {2,10,1869} initializes the member variable date, month and year of the structure variable Gandhi_jay to 2,10,1869 respectively.

If number of elements placed in an initialization is less than the member of members present, then the remaining members are initialized to zero, or NULL.

3.3.5. Accessing and giving values to members

The members themselves are not variables. They should be linked to structure variables in order to make them meaningful members. *To refer to the structure member, member operator ‘.’ is used.* This operator establishes a link between a member and a variable. This operator is also known as “dot operator” or “period Operator”. A link is established by specifying the name of the variable, followed by a period (dot), and then the name of the member. The syntax is

variable_name.member_name

For example price.c is the variable representing the price of c.

scanf function is used to assign values like

```
scanf("%s", c.title);
scanf("%d",& c.pages);
```

variables to the members of book1 can be assigned in the following way

```
strcpy(c.title,"basic");
strcpy(c.author,"Balagurusamy");
c.pages=250;
c.price=28.50;
```

Example:

```
#include< stdio.h >
void main()
struct stu_detail
{
    int id_no;
    char name[20];
    char address[20];
    int age;
}newstudent;
printf("Enter the student information");
printf("Now Enter the student id_no");
scanf("%d",&newstudent.id_no);
printf("Enter the name of the student");
scanf("%s",&new student.name);
printf("Enter the address of the student");
scanf("%s",&new student.address);
printf("Enter the age of the student");
```

```

scanf("%d",&new student.age);
printf("Student information\n");
printf("student id_number=%d\n",newstudent.id_no);
printf("student name=%s\n",newstudent.name);
printf("student Address=%s\n",newstudent.address);
printf("Age of student=%d\n",newstudent.age);
}

```

Structure variables can also be assigned to each other, just like with other variable types:

dob1 = dob2;

Each member of dob1 gets assigned the value of the corresponding member of dob2

The same member names can appear in different structures. There will be no confusion to the compiler because when the member name is used it is prefixed by the name of the structure variable.

3.3.6. Structures within structures

A structure can be declared within another structure. Some times it is required to keep a compound data items within another compound data item. *Structures within a structure is called as nesting of structures.*

Consider the following structure “student”. In this case the structure *date* may be placed inside this structure. The structure ‘date’ contains date of birth of student.

```

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

struct student
{
    char name[20];
    struct date dob;
};

```

The structure “student” contains another structure “date” as its one of its members. Following program demonstrates how to initialize and manipulate the structure student

```

main()
{
    struct student raja ={ "Raja", {14,8,85}};

    printf("\n Name %s",raja.name);
    printf("\n Date of birth: %d - %d - %d", raja.dob.day, raja.dob.month, raja.dob.year);
}

```

Note that the date, month, year are referred with another member operator along with raja.

Output : Name : Raja
 Date of Birth : 14 - 8 - 85

It should be understand that the total bytes allocated to raja is 26. ($20+2+2+2 = 36$)

Syntax for declaring a structure within another structure.

```
struct tag-name1 {declaration 1; declaration 2; ...declaration;};
```

```
struct tag-name2
{
    declaration1;
    decalaration2;
    -----
    struct tag-name1 variale-name1;
    -----
    declaration;
}
```

Nesting of more than one type of structure is permissible. However, a structure cannot be nested within itself.

3.3.7. Arrays of structures

Arrays of structures are commonly used when a large number of similar records are required to be processed together.

Assume, a library has 1000 books. The data related to the above books can be organized in an array of structures. The idea is very simple. First create a structure template. (i.e., declare a structure). Then define an array with this structure for a specified size.

Consider the following example.

```
struct lib_books
{
    char title[20];
    int pages;
};
```

Then array of structure used is *struct lib_books library{1000}*.

In the above statement library is an array containing 1000 elements of the type *struct lib_books*. The advantage is that all these 1000 records can be easily manipulated with the help of loops.

The first record is referred as library[0], second record is referred by library [1] and so on. The first record's title and pages fields are referred as library[0].title and library[0].page.

Hence array of structures is used to manage large number of records easily.

3.3.8. Arrays within structures

A structure can have an array within it. Consider the above example 'lib-books'. Here the first field name is an array which is declared as

```
char title[20];
```

Like this a structure can have number of arrays as members.

Example

```
struct student
{
    char name[20];
    int semester;
    int year;
    char branch[10];
    int mark[3];
};
```

In the above example there are 5 members namely name, semester, year, branch and mark. Out of these 5 fields, name and branch are character arrays and mark is an integer array.

The member mark contains three elements mark[0], mark[1] and mark[2]. If the structure is used to create a variable, then the total amount of memory allocated is 45 bytes ($25+2+2+10+2*3$) .

Now let us see how to refer the various elements of the array mark. Consider the following definition.

```
main()
{
struc student DCT;
-----
```

The first subject mark is referred as DCT. mark[0]. And second subject mark as DCT.mark[1] and so on. This is how arrays can be placed inside structure in order to declare an user defined data type that is representing a complex real world entity.

3.3.9. Unions - Introduction

Union is another compound date type like structure. Union is used to minimize memory utilization. In structure, each member has the separate storage location. But in union, all the members share the common place of memory. Therefore, a union can handle only one member at a time.

Unions are useful for application involving multiple members, where values need not be assigned to all the members at any one time.

3.3.10. Declaration of union

Like structures union can be declared using the keyword *union*. The general format for declaring union is,

```
union tag_name
{
    data_type member_1;
    data_type member_2;
-----
    data_type member_n;
};
```

where union is the key word. tag-name is any user defined data name.

Example

```
union Bio
{
    char name[10];
    int age ;
    float height;
};
```

In the above example ‘union Bio’ has 3 members. First member is a character array ‘name’ having 10 characters (i.e., 10 bytes). Second member is an integer ‘age’ that requires 2 bytes. Third member is a float ‘height’ requiring 4 bytes. All the three members are different data types.

Here all these 3 members are allocated with common memory. They all share the same memory. The compiler allocates a place of storage that is large enough to hold the largest type in the union. In the declaration above, the member “name” requires 10 bytes, which is the largest among the members.

The total memory allotted is also different from structures. In case of structure it will be $10+2+4 = 16$. But here it is only 10.

Union variables are created just like structure variables. For example to create a variable for above ‘union Bio’, the following code is used.

```
main()
{
union Bio studentBio;
-----
```

3.3.11. Initializing Unions

Unions must always be initialized with its first field only. For example to initialize the above ‘Union Bio’ the statement used is

```
main()
{
union Bio studentBio = {"Rama"};
-----
```

Here first element is a character array. Hence the variable studentBio is initialized with “Rama” which is a character constant. The following are erroneous initialization for the ‘union Bio’.

```
union Bio studentBio = {32};
union Bio studentBio = {172.3};
```

Advantages of Union

- It is used to minimize memory utilization.
- It is used to convert data from one type to another type.
- It is used to write a record into a file as a character.

S.No	STRUCTURES	UNIONS
1	Every member has its own memory space	All members use the same memory space
2	Can handle all members as required at a time	Can handle only one member at a time
3	All members can be initialized	Only first member may be initialized
4	Difference Interpretations for the same memory location is not possible	Different Interpretations for the same memory location are possible
5	More storage space required	Conservation of memory is possible

Programs Using Functions, structure aand Unions

1. Write a C program to calculate the circumference and area of a circle given its radius using functions. Implement calculation of circumference and areas as separate functions.

```
#include <stdio.h>
const float pi = 3.141;
float area(float r); //function prototype
float circum(float r); //function prototype
int main(){
    float radius;
    printf("Enter radius: ");
    scanf("%f",&radius); //read radius
    printf("\nArea : %.2f", area(radius));
    printf("\nCircumference : %.2f", circum(radius));
}
float area(float r)
{
    return (pi*r*r);
}
float circum (float r)
{
    return (2*pi*r);
}
```

2. Write a function called ‘prime’ that returns 1 if its argument is a prime no and returns zero otherwise

```
#include<stdio.h>
main()
{
    int i,j,number;
    printf("\nEnter the number:");
    scanf("%d",&number);
    if(prime(number))
        printf("\n\n The given number is prime number");
    else
        printf("\n\n The given number is not prime number");
}
int prime(int n)
{
    int j;
    for(j=2;j<=n-1;j++)
        if(n%j == 0) break;
    if(j==n) return 1;
    else return 0;
}
```

3. Write a function sub-program in ‘C’ to find the value of y.

```
y =x3+2x-10 if x>=10  
= /x/ if x < 0  
= 3x if 0 < x < 10
```

Use it in main program to evaluate y for x varying from -5 to 15 in steps of 0.5.

```
#include<stdio.h>  
#include<math.h>  
main()  
{  
    float x,y;  
    float findy(float x) /* Prototype declaration */  
    clrscr();  
    for(x=-5;x<=15;x+=.5)  
    {  
        printf("\n\n The y value is :%f",findy(x));  
        getch();  
    }  
    getch();  
}  
  
float findy(float x)  
{  
    float t;  
    if(x>=10.0)  
        return((x*x*x)+(2.0*x)-10.0);  
    else if (x > 0)  
        return(3.0*x);  
    else  
    {  
        t=(x* -1) /* for absolute value */  
        return t;  
    }  
}
```

4. Write a program to find the sum of the digits of a given number using function

```
/*find the sum of the digits of a given number using function*/
```

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

```
void main()  
{  
int sumdigit(long int n);/*function prototype*/
```

```

long int num;
int sum;
clrscr();
printf("Enter the number\n\n");
scanf("%ld",&num);
sum=sumdigit(num);/*function call*/
printf("\n\nSum of digits of %d is %d\n",num,sum);
getch();
}

/*function defunition sumdigit()*/
int sumdigit(long int n)
{
int r,s=0;/*local variables*/
do
{
    r=n%10;
    s+=r;
    n/=10;
} while(n!=0);
return(s);
}

```

5. Write a program to find the largest of three numbers using function

```

/*largest of three numbers using function*/
#include<stdio.h>
void main()
{
int max(int x,int y,int z); /*function prototype*/
int a,b,c,result;
printf("Enter the three numbers\n:");
scanf("%d%d%d",&a,&b,&c);
printf("\na=%d b=%d c=%d\n",a,b,c);
result=max(a,b,c);/*function call*/
printf("\n\nLargest of three numbers is %d\n",result);
getch();
}
/*function definition max()*/
int max(int x,int y,int z)
{
int big;/*local variable declaration*/
big=x;
if(y>big)
    big=y;
if(z>big)
    big=z;
return(big); }

```

6. Using recursion, write a C program to reverse a given number.

```
#include<stdio.h>
void main()
{
int n,r;
printf("enter an integer");
scanf("%d",&n);
rev(n);
getch();
}
rev (int n)
{
if (n>0)
{
printf ("%d",n%10);
rev(n/10);
}}
```

7. Devise a structure template that will hold the name of a month, a three-letter abbreviation for the month, the number of days in the month, and the month number.

```
Struct month
{
    char name[10];
    char abbrev[4];
    int days;
    int mo_numb;
};
```

8. Define an array of 12 variables of struct month of the previous question and initialize it for a non-leap year.

```
struct month months[12] = {
    {"January", "jan", 31, 1},
    {"February", "feb", 28, 2},
    {"March", "mar", 31, 3},
    {"April", "apr", 30, 4},
    {"May", "may", 31, 5},
    {"June", "jun", 30, 6},
    {"July", "jul", 31, 7},
    {"August", "aug", 31, 8},
    {"September", "sep", 30, 9},
    {"October", "oct", 31, 10},
    {"November", "nov", 30, 11},
    {"December", "dec", 31, 12} };
```

- 9. Define a structure “cricket” that will contain the following information: player name , country name , batting average. Using “cricket”, declare an array ‘player’ with 50 elements and write a program to read the info about all the 50 players and print a country-wise list containing names of players with their batting average**

```

struct cricket
{
    char p_name[20];
    char country[20];
    float bat_avg;
};

struct cricket player[50];
main()
{
    int i;
    char country[20];
    for(i=0;i<50;i++)
    {
        printf("\n Enter the details about %d batsman",i+1);
        printf("\n Enter the player name:");
        gets(player[i].p_name);
        printf("\n Country name:")
        gets(player[i].country);
        printf("\n batting average:");
        scanf("%f",& player[i].bat_avg );
    }
    printf("\n Enter the country name to print country- wise list:");
    gets(country);
    printf("\n\t\t Country wise list");
    printf("\n Batsman Name \tCountry name\t\tBat.Average");
    for(i=0;i<50;i++)
        if(strcmp(player[i].country,country)== 0)
        { printf("%-20s\t%-20s\t%f",player[i].p_name,
              ,player[i].country,player[i].bat_avg); }
    printf( "\n");
}

```

- 10. Write a program to create a record of 10 countries and capitals using structures. Extend the above program to print the capital if country is given.**

```

#include<stdio.h>
#include<string.h> /* for strcmpl() which ignores case */
struct count_capital
{
    char country[25];
    char capital[25];
} arr[10]; /* array of structures declared */
main()

```

```

{
    int i, total, choice;
    char country_name[25], capital_name[25];
    clrscr();
        printf("\nEnter countries and corresponding capitals :\n", )
    for(i=0;i<10;i++)
    {
        printf("\nCountry : ");
        gets(arr[i].country);
        printf("Capital :");
        gets(arr[i].capital);
    }
    clrscr();
    printf("Enter the option : \n");
    printf("\n1.Display the list \n");
    printf("\n2.Display the inputted country's capital \n");
    printf("\nOption : ");
    scanf("%d", &choice);
    clrscr();
    switch(choice);
    {
        case 1:
            printf("Sr. No.      Country      Capital\n");
            for(i=0;i<20;;i++)
                printf("\n%25s%25s", arr[i].country, arr[i].capital);
            break;
        case 2:
            printf("Enter the country : ");
            gets(country_name);
            for(i=0;i<total;i++)
            {
                if(strcmpi(arr[i].country,country_name)==0)

                {
                    printf("\nCountry : %s\n", arr[i].country);
                    printf("\nCapital : %s\n", arr[i].capital);
                    break;
                }
            }
            if(i==total)
                printf("\nNo match found for : %s\n", country_name);
    }
}

```

- 11. Define a structure for an employee of an organization having employee code, name, address, phone number and number of dependents. Assume that “allEmployees” is an array of employees in ascending order on the employee code. Write a function to display the details of an employee given its employee code**

```

struct employee
{
int emp_code;
char emp_name[30];
char emp_address[50];
char emp_ph_num[10];
int no_of_dep;
}allEmployees[100],b;
void display()
{
int ctr=0;
fp=fopen("employee.c","r");
rewind(fp);
while(fread(&allEmployees,sizeof(allEmployees[i]),1,fp)==1)
{ctr++;
clrscr();
heading();
printf("\n\n\n\tFollowing are the details :-");
printf("\n\n\tRecord #%-d",ctr);
printf("\n\n\tCode : %d",allEmployees[i].emp_code);
printf("\n\n\tName : %s",allEmployees[i].emp_name);
printf("\n\n\tAddress : %s",allEmployees[i].emp_address);
printf("\n\n\tPhoneNumber:%s",allEmployees[i].emp_ph_num);
printf("\n\n\tNumber of Dependents
:%s",allEmployees[i].no_of_dep);
printf("\n\n\n\tPlease Press Enter...");
```

12. Using recursion, write a C program to reverse a given number.

```

#include<stdio.h>
#include<conio.h>
void main(){
int n,r;
printf("enter an integer");
scanf("%d",&n);
rev(n);
getch();}
rev (int n)
{
if (n>0)
{
printf ("%d",n%10);
rev(n/10);}}
```

SUMMARY

- A function is a self-contained program segment that performs a particular task.
- C functions are classified as 1. Pre-defined functions 2. Programmer - defined functions
- Pre-defined functions are already written by compiler developers. Pre-defined functions are not written by the programmer. Pre-defined functions are commonly used in all the programs.
- Programmer defined functions can be written by the programmer at the time of writing a program.
- If type is not specified in the function definition, then the function returns an int value.
- Arguments appearing in the parentheses are called as formal parameters or formal arguments or dummy arguments.
- Information is returned from the function to the calling program through a return statement. A limitation of return statement is that it can return only one value
- Function declaration is the activity that is telling the compiler that a function is defined somewhere below.
- The function main () invokes other functions within it. It is the first function to be called when the program starts execution.
- A statement that invokes another function is called as the function call statement.
- Important points to be considered when calling a function are:
- A semicolon is used at the end of the statement when a function is called.
- Parentheses are compulsory after the function name.
- Whenever a function uses the service of another function, then the former one is called as calling function and the later one is called as called function.
- The process of invoking a function is referred as “Calling a function”.
- While calling the function, some values are provided within (). These values are called as actual parameters or actual arguments.
- This procedure for passing the values of arguments to a function is known call by value
- The scope of variable refers to visibility or accessibility of a variable. The lifetime of a variable refers to the existence of a variable in memory.
- The four classes of a variable, as classified by their scope and life time are 1. Automatic Variables 2. External variable 3. Static variable 4. Register variable.
- Automatic variables are declared inside a function. They are created as soon as a function starts execution, and used within the function. At the end of the execution of the function, it is destroyed.
- Automatic variables are therefore local to the function and are not recognized outside the function. So automatic variables are also called as “local variable” or “internal variable”.
- External variables are active throughout the program execution. They are also known as global variables. External variables can be accessed by any function in the program. External variables are declared outside a function.
- Registers variables are placed in one of the machine registers, instead of using memory. Accessing the register is very fast compared to memory. So frequently used variables are placed in register. This will increase the execution speed

- ceil function returns the smallest integer greater than or equal to the given value. floor function returns the largest integer less than or equal to the given value.
- In C it is possible to call a function itself. Recursion is a process by which a function calls itself repeatedly until some specified condition has been satisfied. A function is called recursive if a statement within the body of function calls the same function.
- There are two types of passing variables to function. They are, call by value and call by reference
- A structure is a derived data type usually representing a collection of variables of same or different data types grouped together under a single name .
- The tag_name is used to declare structure variable anywhere in the program
- The “struct” keyword is used to create a structure variable.
- Defining a structure means creating variables to access the members in the structure.
- An array is a collection of same data types. But structure is a collection of different data types.
- To refer to the structure member, member operator ‘.’ is used. This operator establishes a link between a member and a variable. This operator is also known as “dot operator” or “period Operator”.
- Nested structures are nothing but a structure within a structure is called nested structure.
- If a member of a structure is a pointer to itself, it is a self – referential structure.
- Array of structures are defined as a group of data types stored in a consecutive memory location with a common variable name.
- A pointer to a structure is similar to a pointer to an ordinary variable. It is created in the same way as a pointer to an ordinary variable is created.
- Union is another compound date type like structure. Union is used to minimize memory utilization. In structure, each member has the separate storage location. But in union, all the members share the common place of memory. Therefore, a union can handle only one member at a time.
- Advantages of union are : (i) It is used to minimize memory utilization. (ii) It is used to convert data from one type to another type. (iii) It is used to write a record into a file as a character.

REVIEW QUESTIONS AND PROGRAMS

PART – A (2 Marks)

1. State any two advantages of functions.
2. What is a pre defined function? What is programmer defined function?
3. Write down any four functions available in ctype.h file
4. Write down any four functions available in math.h file
5. Write down the syntax and use of isalpha() function.
6. Write down the syntax and use of islower() function.
7. State the difference between ceil() and floor() functions
8. Give the general form of function definition.
9. What are the purposes of return statement.
10. What are the general forms of return statement.
11. What is a “Calling function”? What is a “Called function”?
12. How the variables are classified by their scope and lifetime?
13. What are external variables? State the other name of external variables.
14. What is register variables? State its use.
15. What is structure ? What is union?
16. Give the general syntax of structure .
17. Give the general syntax of union.

PART – B (3 Marks)

1. With syntax , explain any three functions available in math.h header file.
2. With syntax , explain any three functions available in ctype.h header file.
3. What is a recursive function? What are advantages of recursion?
4. Write the general form of structure. Give an example.
5. Give the example for structure initialization. Give an example.
6. Write down the main differences between an array and a structure.
7. What are the three ways of passing a structure to a function?
8. Define compare union with structure..
9. Write the general form of union. Give an example.

PART – C (5 Marks / 10 Marks)

- 1 What is a function? How the functions are classified? Explain.
- 2 Explain the syntax and usage of any six functions available in math.h header file.
- 3 Explain the syntax and usage of any six functions available in ctype.h header file.
- 4 Explain the syntax and usage of any six functions available in stdio.h header file.
- 5 State any five reasons for using user-defined function.

- 6 Explain the elements of function definition with an example.
- 7 State the use of return statement. What are the different forms of return statement? State the limitations of return statement.
- 8 What do you meant by function prototyping? Explain.
- 9 How you will call a function? Explain your answer by giving an example.
- 10 What is register variable? Give its syntax. Also state the advantages and disadvantages of register variable.
- 11 Compare different types of storage classes.
- 12 What is a recursion? Give an example program.
- 13 What is structure? List down the differences between structure and array.
- 14 How you will define a structure? How you will declare and initialize the values to variable?
- 15 Write down the procedure for accessing a variable in structure. Give an example.
- 16 Explain structures within structures.
- 17 What is union? List down the differences between structure and union.
- 18 How you will define a union? How you will declare and initialize the values to variable?
- 19 Explain call by value with an example program.
- 20 Explain the difference between a function declaration and function definition
- 21 Distinguish between the following: (i) Automatic and static variables (ii) Global and local variables.
- 22 What are the elements of user defined function? Explain in detail.
- 23 How the user defined functions are classified? Explain each function with an example.
- 24 Explain briefly about the scope and lifetime of variables.
- 25 Explain the following; (i) Array of structures and (ii) Arrays within structures.
- 26 Write a C program to calculate the circumference and area of a circle given its radius using functions. Implement calculation of circumference and areas as separate functions.
- 27 Write a function called 'prime' that returns 1 if its argument is a prime no and returns zero otherwise.
- 28 Write a function sub-program in 'C' to find the value of y.

$$Y = \begin{cases} x^3 + 2x - 10 & \text{if } x \geq 10 \\ -x & \text{if } x < 0 \\ 3x & \text{if } 0 < x < 10 \end{cases}$$

Use it in main program to evaluate y for x varying from -5 to 15 in steps of 0.5.
- 29 Write a program to find the sum of the digits of a given number using function
- 30 Write a program to find the largest of three numbers using function

- 31 Using recursion, write a C program to reverse a given number.
- 32 Using recursion, write a C program to reverse a given number.
- 33 Devise a structure template that will hold the name of a month, a three-letter abbreviation for the month, the number of days in the month, and the month number.
- 34 Define an array of 12 variables of struct month of the previous question and initialize it for a non-leap year.
- 35 Define a structure “cricket” that will contain the following information: player name , country name , batting average. Using “cricket”, declare an array ‘player’ with 50 elements and write a program to read the info about all the 50 players and print a country-wise list containing names of players with their batting average
Write a program to create a record of 10 countries and capitals using structures. Extend the above program to print the capital if country is given.
- 36 Define a structure for an employee of an organization having employee code, name, address, phone number and number of dependents. Assume that “allEmployees” is an array of employees in ascending order on the employee code. Write a function to display the details of an employee given its employee code.

OBJECTIVES

At the end of the unit, the students will be able to

- Define pointer
- Discuss the advantages of pointers.
- Understand the concept of pointers
- Learn the use of pointer variables.
- Understand pointer arithmetic.
- Learn about the concept and construction of array of pointers and pointer to array.
- Discuss pointers and functions
- Get an idea about pointers to functions – functions as arguments to another function
- Understand dynamic memory location using pointers
- Describe about the functions used in dynamic memory allocation.
- Differentiate dynamic memory functions.

INTRODUCTION

Pointer is an important feature of C Language. A pointer is a derived data type in C. It is powerful and handy tools of C language. As the pointers are closely associated with the arrays, they provide an alternative way to access the individual element of an array. C Programmers use pointers to make the code more efficient. Some C programming tasks are performed more easily with pointers, and other tasks, such as dynamic memory allocation, cannot be performed without using pointers. So it becomes necessary to learn pointers to become a perfect C programmer.

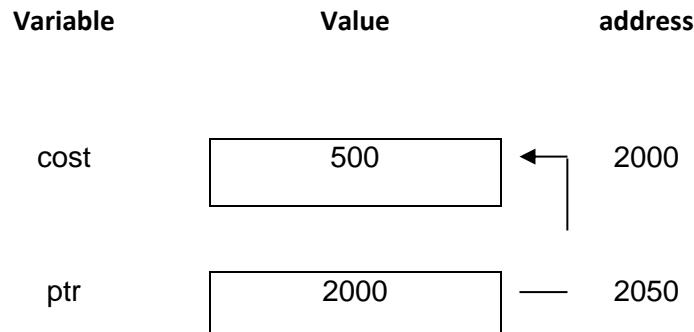
C language requires the number of elements in an array to be specified at compile time. But sometimes, it is not possible to judge the number of elements. C languages permit a programmer to specify an array's size at run time. C language have the ability to calculate and assign, during execution, the memory space required by the variables in a program. The process of allocating memory during run time is called as dynamic memory allocation.

4.1. POINTERS**4.1.1. DEFINITION**

A pointer is a variable that is used to store the address of another variable. Since a pointer is also another variable, its value is also stored in the memory in another location. The pointer variable might be belonging to any of the data type such as int, float, char, double, short etc.

Each memory cell in the computer has an address that can be used to access that location. So a pointer variable points to a memory location. The contents of this memory location can be accessed and changed by using pointer variable.

Assume a variable “cost”. Then the address of this cost is assigned to pointer variable ptr. The link between the variables ptr and cost is shown in figure. Assume the address of ptr is 2050. The link between the two variables are shown in the following figure.



POINTERS CONTAIN MEMORY ADDRESSES, NOT DATA VALUES!

4.1.2. ADVANTAGES OF POINTERS

1. Pointers are used to increase the speed of the execution.
2. Pointers are used to reduce the length and complexity of program.
3. A pointer is used to access a variable that is defined outside the function.
4. Storage space is saved when using pointer array to character strings.
5. Pointers can also be used in efficient manner to access the individual array elements.

4.1.3. DEFINING A POINTER VARIABLE

Pointer variables, like all other variables, must be declared before they may be used in a C program. The interpretation of a pointer declaration is somewhat different than the interpretation of other variable declarations. For defining a pointer variable ‘*’ symbol is used. A pointer variable must be declared with preceding the variable name. This signifies that it is not an ordinary variable but a pointer variable. The type of data it is pointing must be specified. The general structure for declaring a pointer variable is

```
data_type *ptr_name;
```

For example, the statement `int *p` declares p as a pointer variable that points to an integer data type.

```
{
```

For example,

```
float a,b;  
float *pb;
```

The first line declares a and b to be floating-point variables. The second line declares pb to be a pointer variable whose object is a floating-point quantity i.e. 'pb' point to a floating point quantity. 'pb' represents an address, not a floating-point quantity. Within a variable declaration, a pointer variable can be initialized by assigning it the address of another variable.

4.1.4. ADDRESS OPERATOR – ACCESSING THE ADDRESS OF A VARIABLE

The address of a variable is obtained with the help of “&” operator at any time. *& operator is called as “address operator”.* The operator & immediately preceding a variable returns the address of the variable. For example, the statement

```
ptr=&num;
```

places the address of num into the variable ptr. If num is stored in memory 21260 address, then the variable ptr has the value 21260.

```
/* A program to illustrate pointer declaration*/
main()
{
    int *ptr;
    int sum;
    sum=45;
    ptr= &sum;
    printf ("\n Sum is %d\n", sum);
    printf ("\n The sum pointer is %d", ptr);
}
```

4.1.5. ACCESSING A VARIABLE THROUGH ITS POINTER

The content of any pointer variable can be accessed with the help of “*” operator. *The operator “*” is known as indirection operator.* If “p” is an pointer variable, then *p is used to access the content of the pointer variable “p”. For example the statement t=*p is used to assign the contents of the pointer variable p to t.

```
/* Program to display the contents of the variable using pointer variable*/
#include<stdio.h>
{
int num, *intptr;
float x, *floptr;
char ch, *cptr;
num=123;
x=12.34;
ch='a';
intptr=&x;
cptr=&ch;
floptr=&x;
printf("Num %d stored at address %u\n",*intptr,intptr);
printf("Value %f stored at address %u\n",*floptr,floptr);
printf("Character %c stored at address %u\n",*cptr,cptr);
}
```

4.1.6. INITIALIZATION OF POINTERS

Before using the pointer, it should be initialized. Static local pointer variables and global pointer variables are initialized with NULL by default. “NULL” is a pointer constant whose numerical value is zero.

Automatic pointer variable can be either initialized with NULL or with address of some other variable that is already defined.

For example in the statement,

```
p = & cost;
```

p contains the address of cost. This is called pointer initialization.

A pointer variable can also be initialized in the declaration statement itself. For example.

```
int a; *b = &a;      is also valid.
```

Example :

```
# include <stdio.h>
main ( )
{
int a=3;
int b;
int *pa; /* to an integer */
int *pb; /* pointer to an integer */
pa=&a; /* assign address of a to pa */
b=*pa; /* assign value of a to b */
pb=&b; /* assign address of b to pb */
printf ("\n% d %d %d %d", a, &a, pa, *pa);
printf ("\n% d %d %d %d", b, &b, pb, *pb);
}
```

The output is : 3 F8E F8E 3

3 F8C F8C 3

The unary operators & and * are members of the same precedence group. The address operator (&) must act upon operands associated with unique addresses, such as ordinary variables or single array elements. Thus, the address operator cannot act upon arithmetic expression, such as $2*(a+b)$. The indirection operator (*) can only act upon operands that are pointers.

Example:

```
# include <stdio.h>
main()
{
int i=3, * j;k; /* j is a pointer to integer */
j=&i; /* j points to the location of i */
k=*j ; /* assign to k the value pointed to by j */
*j=4 /assign 4 to the location pointed to by j
printf("i=%d, *j=%d, k=%d\n", i, *j, k);
}
```

The output is as follows: i=4, *j=4, k=3

4.1.7. NULL POINTERS

It is always a good practice to assign a NULL value to a pointer variable in case you do not have an exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a **null** pointer.

The NULL pointer is a constant with a value of zero defined in several standard libraries. Consider the following program –

```
#include <stdio.h>
int main () {
    int *ptr = NULL;
    printf("The value of ptr is : %x\n", ptr );
    return 0;
}
```

When the above code is compiled and executed, it produces the following result –

The value of ptr is 0

In most of the operating systems, programs are not permitted to access memory at address 0 because that memory is reserved by the operating system. However, the memory address 0 has special significance; it signals that the pointer is not intended to point to an accessible memory location. But by convention, if a pointer contains the null (zero) value, it is assumed to point to nothing.

To check for a null pointer, you can use an 'if' statement as follows –

```
if(ptr) /* succeeds if p is not null */
if(!ptr) /* succeeds if p is null */
```

4.1.8. POINTER EXPRESSIONS

Pointer variables can also be used in expressions. Assume a and c are pointers. Then the following is an example of pointer expression

$$c = *a + 14;$$

The above expression adds the value 14 to the contents of address stored in a and assigns the new value to c. The following are also valid expressions.

2. $x = *a * *b;$
3. $total = total - *a;$
4. $z = *b/*a + 10;$
5. $*b = *b + 10;$

The pointers may not be used in division or multiplication. For example $*a/3$ or $*a \times 3$ are not allowed.

Two pointer variables can be compared provided both points to objects of same type. Some examples are $a > b$, $a == b$, and $a != b$. Comparisons can be used in handling arrays and strings.

4.1.9. INCREMENT AND SCALE FACTOR

The content of location a is incremented by 1 through a+=1; The contents can also be incremented either through ++*p; or (*p)++; It is also possible to add integers or subtract integers from pointers. Subtraction of one pointer from another pointer is also possible. For example the following statements are valid.

a++ - -b; a-b;

When a pointer is incremented, its value is incremented by the length of the data type it points to. For example assume a is an integer pointer and its value is 5000. After performing the operation a = a+1, the value of a will be 5002, and not 5001. Because, integer value occupies 2 bytes. This length is called the *scale factor*.

#include<stdio.h> main() { int i = 3, *x ; float j = 1.5, *y ; char k = 'c',*z ; x=&i ; y=&j ; z =&k ; printf ("Original value in x = %d\n", x) ; printf ("Original value in y = %d\n", y) ; printf ("Original value in z = %d\n\n", z) ; x++; y++; z++; printf ("New value in x = %d\n", x) ; printf ("New value in y = %d\n", y) ; printf ("New value in z = %d\n", z) ; getch(); }	Output e i, j and k are stored in memory at addresses 1002, 2004 and 5006, the output would be... Value of i = 3 Value of j = 1.5 Value of k= c Original value in x = 1002 Original value in y = 2004 Original value in z = 5006 New value in x = 1004 New value in y = 2008 New value in z = 5007
---	--

4.1.10. KEYPOINTS TO REMEMBER ABOUT POINTERS IN C

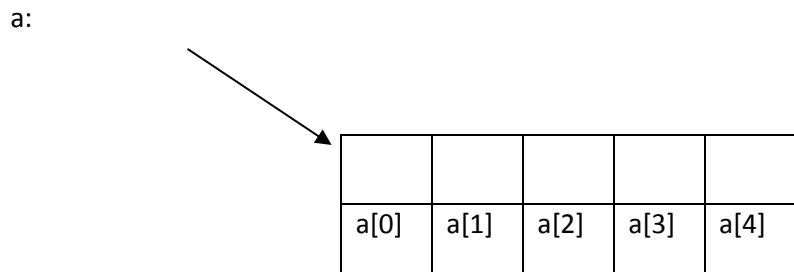
1. A pointer variable can be assigned the address of an ordinary variable.
2. A pointer variable can be assigned the value of another pointer variable provided both pointers point to objects of the same data type.
3. An integer quantity can be added to or subtracted from a pointer variable.
4. A pointer variable can be assigned a null (zero) value.
5. One pointer variable can be subtracted from another provided both pointers point to elements of the same array.
6. Two pointer variables can be compared provided both pointers point to objects of the same data type.
7. A pointer variable can be assigned to address of another variable.
8. A pointer variable can be assigned the values of another pointer variables.

9. A pointer variable can be initialized with NULL or 0 value.
10. A pointer variable can be prefixed or postfixed with increment and decrement operator.
11. An integer value may be added or subtracted from a pointer variable.
12. When 2 pointers points to the same array, one pointer variable can be subtracted from another.
13. When two pointers points to the objects of save data types, they can be compared using relational opearotor.
14. A pointer variable cannot be multiple by a constant.
15. Two pointer variables cannot be added.
16. A value cannot be assigned to an arbitrary address

4.1.11. POINTERS AND ARRAYS

Pointers and arrays have got close relationship. In fact array name itself is a pointer. Some times pointers and array names can be interchangeably used.

The array name always points to the first element of the array. For example, an array `int a[5];` is shown in the following figure.



Here 'a' points to the first element of the array. That means a is `&a[0]`.

In arrays `a` and `&a` both are same. Hence `a`, `&a` and `&a[0]` all means same. But there exists difference between array and pointer in certain aspects. . Similarly, the address of the second array element can be written as either `&a[1]` or as `(a+1)` and so on.

Since `&x[i]` and `(x+i)` both represent the address of the ith element of x then `x[i]` and `*(x+i)` both represent the contents of that address. The two terms are interchangeable.

Array name is a constant pointer. It can't be assigned with new address. It always points to the first element. But it's address can be used for some other purpose.

Example

```
main()
{
int a[5];
int i;
a++;      /* error*/
a= &i;      /* error*/
*(a+2) = 30; /* ok*/
```

a:

A diagram illustrating a pointer. A black arrow points from the variable 'a' to the third cell of a 2x5 grid. The grid has two rows. The top row contains five empty cells. The bottom row is labeled 'a[0]' through 'a[4]' under each cell respectively. The third cell in the bottom row contains the value '30'.

		30		
a[0]	a[1]	a[2]	a[3]	a[4]

Here *(a+2) actually refers to a [2]. At some places pointers and arrays can be interchangeably used.

A program to display the contents of array using pointer

```
main()
{
int a[100];

int *ptr;
int i,j,n;
printf("\nEnter the elements of the array\n");
scanf("%d",&n);
printf("Enter the array elements");
for(i=0;i< n;i++)
scanf("%d",&a[i]);
printf("Array element are");
for(ptr=a,ptr< (a+n);ptr++)
printf("Value of a[%d]=%d stored at address %u",j+=,*ptr,ptr);
}
```

A Program to compute the sum of all the elements in an array.

```
main()
{
int *p, total = 0; i=0;
int a[5]={2,3,4,6,7};
p = a;
for (i=0; i<5; i++)
{
    total = total + *p;
    p++;
}
printf("\nTotal%d", total)
}
```

4.1.12 .POINTERS AND FUNCTIONS

A function also has an address location in memory. It is therefore possible to declare a pointer to a function which can be used as an argument in another function.

A pointer to a function is described as follows

```
type (*pointer-name) ( );
```

For example `int (*sqrt) ()` tells the compiler that `sqrt` is a pointer to a function which returns an int value.

For example, in the statements

```
int *sqr (), square ();
sqr = square;
```

`sqr` is pointer to a function and `sqr` points to the function `square`. To call the function `square`, the pointer `sqr` is used with the list of parenthesis. That is,

```
(*sqr) (a,b)
```

Is equivalent to

```
square (a,b)
```

Pointers as functions arguments (Call by Reference)

A pointer plays an important role when used with functions. Pointer variables can be passed as arguments to some other function. That is, the address of a variable is passed to another function. This type of function call is known as *call by reference*. The function is allowed to access and change the variable of a calling function. When passing the address of a variable as an argument to a function, the parameters receiving the address should be pointers.

When the function invocation uses the method call by value, the value of actual parameter is copied to dummy parameters.

In the case of call by reference method, instead of passing the values, the addresses of actual arguments in the calling function are copied into formal arguments of the called function. Hence it is possible to change the actual argument within the function body. This means, using these addresses it is possible to have an access to the actual arguments and hence manipulate them.

Since the actual argument variable and the corresponding dummy pointer refer to the same memory location, changing the contents of the dummy pointer will- by necessity- change the contents of the actual argument variable. The following program illustrates this fact.

```
#include <stdio.h>
main ()
{
int i,j;
i=2;
j=5;
printf("i=%d and j=%d\n", i,j);
/* Now exchange them */
swap(&i,&j);
printf("\nnow i=%d and j=%d\n", i,j);
}
swap(i,j)
int *i,*j;
{
int temp=*i; /* create temp and store into it the value pointed to by "i"*/
*i=*j; /*The value pointed to by j is stored in the location pointed to by i*/
```

```
*j=temp; /* Assign temp to the location pointed to by j*/  
}
```

The output is as follows:

i=2 and j=5

Now i=5 and j=2

If the formal parameters i and j of the swap function were declared merely as integers, and the main function passed only i and j rather than their addresses, the exchange made in swap would have no effect on the variables i and j in main. The variable temp in swap function is of type int, not int*. The values being exchanged are not the pointers, but the integers being pointed to by them. Also temp is initialized immediately upon declaration to the value pointed to by i.

C program to find the highest of three numbers using pointer to function is listed below:

```
#include<conio.h>  
void main()  
{  
int x,y,z;  
clrscr();  
printf("\n Enter three numbers : ");  
scanf("%d %d %d",&x,&y,&z);  
printf("\n\n The highest of the three numbers is : ");  
highest(&x,&y,&z);  
getch();  
}  
highest(a,b,c)  
int *a,*b,*c;  
{  
if(*a > *b && *a > *c)  
printf("%d",*a);  
else if(*b > *a && *b > *c)  
printf("%d",*b);  
else  
printf("%d",*c);  
}
```

4.1.13 .POINTERS AND CHARACTER STRINGS

Character array is called as string variable. Pointer is used to access the individual character in a string.

Example:

```
main()  
{  
char name[30];  
char *p;  
p = name;
```

```
strcpy(p,""salem"");
printf("\n%s", name);
}
```

Output : salem

Here ‘p’ is an alias name of character array ‘name’. Character pointers can be used effectively for various purposes.

Example:

```
main()
{
char name[30] ={"salem"};
char name1[30];
void mycopy ( char *, char *);
mycopy(name1,name);
printf("\n%s", name1);
}

void mycopy ( char *d, char *s)
{
    While ( *s !='\0')
    {
        (*d++ = *s++);
    }
}
```

Output salem

The above program copies one string to another string. In the above function pointer arithmetic is used. After copying the content of one pointer to another, they are incremented to point the next element. Hence next element is copied. This is repeated until ‘\0’ is encountered. The while loop will be continued as long as the condition is true i.e., non-zero.

4.1.14 .ARRAY OF POINTERS TO STRINGS

Sometimes it is required to store more than one string in an array. One way is to use two dimensional character array.

Example

```
main()
{
int i;
char name[3][12]={"Ramu", "Govindan", "Rajasekaran"};
for (i=1; i<3; i++)
{
    printf("\n%s      ", name[i]);
}}
```

Note that irrespective of the size of the names totally $3 * 12 = 36$ bytes of memory is allocated. When using array of pointers , the memory usage can be minimized.

Example

```
main()
{
int i;
char *name[3]={"Ramu", "Govindan", "Rajasekaran"};
for (i=1; i<3; i++)
{
    printf("\n%s      ", name[i]);
}
}
```

In this declaration name[] is an array of pointers. It contains base addresses of respective names. That is, base address of “Ramu” is stored in name[0], base address of “Govindan” is stored in name[1] and so on.

Here only required amount of memory is allocated. i.e., $5 + 9 + 12 = 26$ bytes. Hence we have saved 10 bytes.

Advantages of storing strings in an array of pointers:

1. One reason to store strings in an array of pointers is to make a more efficient use of available memory.
2. Another reason to use an array of pointers to store strings is to obtain greater ease in manipulation of the strings.

Limitation of array of pointers to strings

When using a two-dimensional array of characters, it is possible to initialize the strings when declaring the array or the strings can be received through **scanf()** function.

However, when using an array of pointers to strings, it is possible to initialize the strings at the place where we are declaring the array, but it is not possible to receive the strings from keyboard using **scanf()**. Thus, the following program would never work out.

```
main( )
{
char *names[6];
int i;
for ( i = 0 ; i <= 5 ; i++ )
{
    printf ( "\nEnter name " );
    scanf ( "%s", names[i] );
}}
```

4.1.15 .POINTERS AND STRUCTURES

In an array, the name of an array stands for the address of its zeroth element. Similarly struct variable represents the address of its first element.

Example

```
typedef struct Bio
{
    char name[30];
    int age;
};

main()
{
    Bio Record;
    Bio *p;
    p = & Record;
    printf( "\n Enter Name;");
    scanf( "%s", (*p).name);
    printf( "\n Enter Age;");
    scanf( "%d", &(*p).age);
    printf( "\n Name;", (*p).name);
    printf( "\n Age%d", (*p).age);
}
```

Output:

Enter Name : Ramu

Enter Age : 28

Name : Ramu

Age : 28

Here in this program first ‘Bio’ struct type is declared. Then in main(), Record of type ‘Bio’ and a pointer ‘p’ of type ‘Bio’ is defined.

Then ‘p’ is made to point ‘Record’. Hence ‘*p’ is an alias name of ‘Record’. There after whenever ‘*p’ is referred, it should be interpreted as ‘Record’. Then the fields of Record name and age are read and they are printed.

Note for referring the fields, use both indirection operation ‘*’ and member operator “.”.

i.e., (*p). name

Parenthesis is required since priority of ‘.’ operator is higher than ‘*’ operator. Apply ‘*’ operator first. Hence change the order of priority by using parenthesis.

This leads to confusion. Hence instead of using ‘*’, ‘.’ and parenthesis, the above is simplified by using the -> operator. The symbol -> is called as *arrow operator*.

Instead of writing `(*p). name`, we write

```
p-> name ; /* no space between p->name */
```

which is more simple.

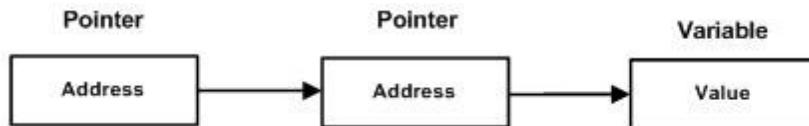
A pointer pointing to a structure just the same way a pointer pointing to an int, such pointers are known as structure pointers. For example consider the following example:

```
#include<stdio.h>
#include<conio.h>
struct student
{
    char name[20];
    int roll_no;
};
void main()
{
    struct student stu[3],*ptr;
    clrscr();
    printf("\n Enter data\n");
    for(ptr=stu;ptr<stu+3;ptr++)
    { printf("Name");
        scanf("%s",ptr->name);
        printf("roll_no");
        scanf("%d",&ptr->roll_no);
    }
    printf("\nStudent Data\n\n");
    ptr=stu;
    while(ptr<stu+3)
    {
        printf("%s %5d\n",ptr->name,ptr->roll_no); ptr++;
    }
    getch();}
```

Here `ptr` is a structure pointer not a structure variable and dot operator requires a structure variable on its left. C provides arrow operator “`->`” to refer to structure elements. “`ptr=stu`” would assign the address of the zeroth element of `stu` to `ptr`. Its members can be accessed by statement like “`ptr->name`”. When the pointer `ptr` is incremented by one, it is made to point to the next record, that is `stu[1]` and so on.

4.1.16 .POINTER TO POINTER

A pointer to a pointer is a form of **multiple indirection**, or a chain of pointers. Normally, a pointer contains the address of a variable. When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below.



A variable that is a pointer to a pointer must be declared as such. This is done by placing an additional asterisk in front of its name. For example, following is the declaration to declare a pointer to a pointer of type int:

```
int **var;
```

When a target value is indirectly pointed to by a pointer to a pointer, accessing that value requires that the asterisk operator be applied twice, as is shown below in the example:

```
#include <stdio.h>
int main ()
{
    int var;
    int *ptr;
    int **pptr;
    var = 3000; /* take the address of var */
    ptr = &var;
    /* take the address of ptr using address of operator & */
    pptr = &ptr;
    /* take the value using pptr */
    printf("Value of var = %d\n", var );
    printf("Value available at *ptr = %d\n", *ptr );
    printf("Value available at **pptr = %d\n", **pptr);
    return 0;
}
```

When the above code is compiled and executed, it produces following result:

```
Value of var = 3000
```

```
Value available at *ptr = 3000
```

```
Value available at **pptr = 3000
```

4.2 DYNAMIC MEMORY ALLOCATION

The process of allocating memory at run time is known as dynamic memory allocation. Dynamic memory management technique is used to optimize the use of storage space. These techniques are used to allocate additional memory space or to release the unwanted space at run time. Using dynamic memory management techniques, the programmer can allocate memory whenever he decides and releases it after using the memory.

The functions used in the dynamic memory management are (i) malloc () (ii) calloc() (iii) realloc() and (iv) free ().

4.2.1 malloc()

The name malloc stands for "memory allocation". This function is used to allocate a block of memory. The required amount of byte should be specified as argument to the function. After allocating memory in the heap (free memory) the function returns the starting address of the block of memory allotted.

Syntax

```
ptr = (cast_type *) malloc (size);
```

where ptr is a pointer of type cast_type.

For example the statement p = (int*) malloc(100); reserves 100 bytes of the memory and the address of the first byte of the memory allocated is assigned to the pointer p of type int.

Example

```
main()
{
    int *p;
    p = (int*) malloc(6);
    p[0] = 10;
    p[1] = 20;
    p[2] = 30;
    printf("\n%d %d %d", p[0] , p[1] , p[2]);
    free (p);
}
```

Output: 10 20 30

Here 'p' is an integer pointer. A pointer variable stores the memory address.

The malloc() allocates 6 bytes in heap and the address is converted to integer address by means of (int *) cast operator. This converted address is assigned to the pointer ‘p’.

Now ‘p’ points to 6 byte of memory block each 2 byte pair representing an integer. In short ‘p’ is an array having 3 elements namely p[0], p[1] and p[2].

There after ‘p’ can be used as if it is an array. But the difference is that whenever required, the elements can be created and after having used it may be released using free(). The following figure shows this setup.

1	0	2	0	3	0
p[0]		p[1]		p[2]	

The storage space allocated dynamically has no name and therefore its contents can be accessed only through a pointer.

Example:

```
// Program to demonstrate usage of malloc( )
#include<stdio.h>
main( )
{
    int n, avg, i, *p, sum = 0;
    printf("Enter the number of students");
    scanf("%d",&n);
    p = (int *) malloc (n * 2);
    if( p == NULL)
    { printf("Memory allocation unsuccesful");

        exit( );
    }
    for( i = 0; i < n; i++)
        scanf("%d",(p + i));

    for( i = 0; i < n; i++)
        sum = sum + *p;

    avg = sum / n;
    printf("Average marks = %d", avg);
    getch();
}
```

The above program asks for the number of students whose marks are to be entered and then allocate only as much memory as is really required to store these marks. The allocation job is done by malloc(). It returns a NULL if memory allocation is unsuccessful. If successful it returns the address of the memory chunk that was allocated.

This address we collected in an integer pointer p. The expression (int *) is used to type cast the address being returned as the address of an integer rather than a character. This type casting is necessary since malloc() by default returns a pointer to a void.

4.2.2 free()

This function is used to deallocate the memory. The release of storage space is important when the storage is limited. When data stored in a block of memory is not needed, then release that portion of the memory.

Syntax

```
free(pointer  
variable);
```

where pointer variable is a pointer to a memory block which has already been created by malloc or calloc.

Example

```
main()  
{  
    int *p;  
    p = ( int*) malloc(6);  
    -----  
    free (p);  
}
```

4.2.3 calloc()

The name calloc stands for "contiguous allocation" malloc() is used to allocate single block of memory. Whereas calloc() will allocate multiple blocks of storage, each of the same size and initialize all the bytes of the memory to zero. calloc function is used when requesting memory space at run time for storing derived data types such as arrays and structure.

Syntax

```
p = (cast_type *) calloc (number of blocks, block size);
```

This allocates continuously memory blocks each with the specified size. The returned address will be the address of the starting byte. If the required amount of memory is not available, then NULL value is returned.

Example

```
main()  
{  
    int *p;  
    p = (int*) calloc(10, sizeof(int));  
    -----  
}
```

This allocates 10 continuous blocks of memory and each block having 2 bytes.

Differences between malloc() and calloc()

Another minor difference between malloc() and calloc() is that by default the memory allocated by malloc() contains garbage values whereas that allocated by calloc() contains all zeros.

While malloc allocates a single block of storage space, calloc allocates multiple block of storage, each of the same size, and then sets all bytes to zero.

4.2.4 realloc()

This function is used to change the memory size previously allocated. The space may be increased or decreased.

Syntax

```
ptr= realloc (pointer variable name, new size);
```

This function allocates a new memory space of size equal to the new size to the pointer variable ptr and returns a pointer to the first byte of the new memory block.

The new size may be large or smaller than the original size. Consider the following example.

```
main()
{
int *p;
p = (int*) malloc(10);
-----
-----
P= (int*) realloc(p,20);
}
```

Here initially 'p' is allotted with 10 bytes. But later it is felt that amount of memory is to be increased. That is done with the help of realloc() function. This time 20 bytes are allotted.

Like that, it is also possible to decrease the memory size.

Here also if it is not possible to allocate memory space, realloc() returns NULL value, and the original block is lost. Here, *ptr* is reallocated with size of newsize.

```
#include <stdio.h>
#include <stdlib.h>
int main(){
    int *ptr,i,n1,n2;
```

```

printf("Enter size of array: ");
scanf("%d",&n1);
ptr=(int*)malloc(n1*sizeof(int));
printf("Address of previously allocated memory: ");
for(i=0;i<n1;++i)
    printf("\n%u\t",ptr+i);
printf("\nEnter new size of array: ");
scanf("%d",&n2);
ptr=realloc(ptr,n2);
for(i=0;i<n2;++i)
    printf("\n%u\t",ptr+i);
return 0;
}

```

PROGRAMS USING POINTERS

1. Consider the following:

P1 is an integer pointer

P2 is a long integer pointer

P3 is a character type pointer

The initial value of P1 is 2800, P2 is 1411 and P3 is 1201.

What is the new value of P1 after P1=P1+1, P2 after P2=P2+1 and

P3 after P3=P3+1; (4)

The initial value of P1 is 2800 which is an integer pointer so new value of P1 is 2802 after P1=P1+1

The initial value of P2 is 1411 which is a long integer pointer so new value of P1 is 1415 after P2=P2+1 because long takes 4 bytes of memory.

The initial value of P3 is 1201 which is a char type pointer so new value of P3 is 1202 after P3=P3+1

2. Write a program to read in an array of integers .Instead of using Subscripting ,however, employ an integer pointer that using to the element currently being read in and which is incremented each time. Write a function to display the array.

```

#include<stdio.h>
#include<conio.h>
#define MAX 50
main()
{

```

```

int array[MAX];
int i,n;
int *employ=array;
clrscr();
printf("\n How many numbers:");
scanf("%d",n);
for(i=0;i<n;i++)
{
    printf("\n Enter the %d number:",i+1);
    scanf("%d",employ+i);
}
display(employ,n)
}
display(int *ptr,int size)
{
    int i;
    for(i=0;i<size;i++)
        printf("\n The %d number : %d",i+1,*(ptr+i));
}

```

3. Write a 'c' program to sort an array of integers using pointers

```

#include<stdio.h>
#include<conio.h>
main()
{
    int number[50];
    int i,j,n,*temp,*ptr=number;
    clrscr();
    printf("\n How many numbers:");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        printf("\n Enter the %d number:",i+1);
        scanf("%d",ptr+i);
    }
    for(i=0;i<n-1;i++)
        for(j=i;j<n;j++)
        {
            if(*(ptr+i)>*(ptr+j))
            {

```

```

        *temp=*(ptr+i);
        *(ptr+i)=*(ptr+j);
        *(ptr+j)=*temp;
    }
}

printf("\n\n The sorted array :");
for(i=0;i<n;i++)
    printf("\n The %d number : %d",i+1,*(ptr+i));
getch();
}

```

4. What is the output of the following?

```

main()
{
    int i=4,j=2;
    jack(&i,j);
    printf("%d%d",i,j);
}
jack(int *i,int j)
{
    *i=*i * *i;
    j=j * j;
}

```

Output 16 2

5. Write a program using pointers to read an array of integers and prints its elements in reverse order.

```

#include<stdio.h>
main()
{
    int array[50],*ptr=array,n,i;
    clrscr();
    printf("\n How many integers:");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        printf("\n Enter the %d number:",i+1);
        scanf("%d",ptr++);
    }
}

```

```

ptr--;
while(ptr>ptr-array)
    printf("\n %d",*(ptr--));
}

```

6. Write a program using pointers, which accepts a string and displays the length of the string

```

#include<stdio.h>
main()
{
    char str[50],*ptr=str;
    int len=0;
    clrscr();
    printf("\n Enter the string:");
    gets(str);
    while(*(ptr++))
        ++len;
    printf("\n\n The length of the string is:%d",len);
}

```

7. Write a program that accepts two strings, compares them and displays which is bigger. Use pointer notations.

```

#include<stdio.h>
main()
{
char str1[50],str2[50];
    char *ptr1=str1,*ptr2=str2;
    int len1=0,len2=0;
    clrscr();
    printf("\n Enter the first string:");
    gets(str1);
    printf("\n Enter the second string:");
    gets(str2);
        while(*ptr1++)
    { ++len1; }

    while(*ptr2++)
    { ++len2 }

    if(len1>len2) printf("\n The first string is bigger");
    else if(len1<len2) printf("\n The second string is bigger");
    else printf("\n The two strings are equal");
}

```

8. Write a program that , given two strings, appends the second string to the first

```
#include<stdio.h>
#include<string.h>
main()
{
    char str1[50],str2[50];
    char *ptr1=str,*ptr2=str2;
    clrscr();
    printf("\n Enter the first string:");
    gets(str1);
    printf("\n Enter the second string:");
    gets(str2);
    ptr1=str1+strlen(str1);

    while(*(ptr++) = *(ptr2++));
    *(ptr1) = NULL;
    printf("\n\n The concatenated string is :%s",str1);
}
```

9. Write a program to reverse a string using pointer.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int i,j;
    char *a;
    clrscr();
    printf("enter a string");
    scanf("%s",a);
    i=0;
    while(*(a+i)!='\0')
        i++;
    for(j=i-1;j>=0;j--)
        printf("%c",*(a+j));
    getch();
}
```

10. Write a C program to display the contents of an array using a pointer arithmetic:

```
#include<conio.h>
void main()
{
    int *p,sum,i;
```

```

static int x[5] = {5,9,6,3,7};
i=0;
p=x;
sum=0;
clrscr();
printf("\nElement Value Address\n\n");
while(i<5)
{
printf(" x[%d] %d %u\n",i,*p,p);
sum+=*p;
i++;
*p++;
}
printf("\n Sum = %d\n",sum);
printf("\n &x[0] = %u\n",&x[0]);
printf("\n p = %u\n",p);
getch();
}

```

11. Differentiate between pointer (*) and address (&) operator using examples.

The indirection operator (*) gets the value stored in the memory location whose address is stored in a pointer variable. The address of (&) operator returns the address of the memory location in which the variable is stored. The output of the following example shows the difference between * and &.

```

#include<conio.h>
void main()
{
int k;
int *ptr;
clrscr();
k=10;
ptr=&k;
printf("\n Value of k is %d\n\n",k);
printf("%d is stored at addr %u\n",k,&k);
printf("%d is stored at addr %u\n",*ptr,ptr);
*ptr=25;
printf("\n Now k = %d\n",k);
getch(); }

```

12. Compare Indirection and Address-Of Operators

Indirection: * is applied to a pointer variable, to refer to the location whose address is stored inside the pointer variable.

- It CANNOT be applied to non-pointer variables.
- It CAN appear on either side of an assignment statement.

Address-Of: & is applied to a non-pointer variable, to return the address of the variable.

- It CAN be applied to a pointer variable.
- It CANNOT appear on the lefthand side of an assignment statement. (You can't change the address of a variable.)

13. Define a structure for an employee of an organization having employee code, name, address, phone number and number of dependents. Assume that "allEmployees" is an array of employees in ascending order on the employee code. Write a function to display the details of an employee given its employee code

```
struct employee
{
    int emp_code;
    char emp_name[30];
    char emp_address[50];
    char emp_ph_num[10];
    int no_of_dep;
}allEmployees[100],b;
void display()
{
    int ctr=0;
    fp=fopen("employee.c","r");
    rewind(fp);
    while(fread(&allEmployees,sizeof(allEmployees[i]),1,fp)==1)
    {
        ctr++;
        clrscr();
        heading();
        printf("\n\n\n\tFollowing are the details :-");
        printf("\n\n\tRecord #%-d",ctr);
        printf("\n\n\tCode : %d",allEmployees[i].emp_code);
        printf("\n\n\tName : %s",allEmployees[i].emp_name);
        printf("\n\n\tAddress : %s",allEmployees[i].emp_address);
        printf("\n\n\tPhoneNumber:%s",allEmployees[i].emp_ph_num);
        printf("\n\n\tNumber of Dependents
        :%s",allEmployees[i].no_of_dep);
        printf("\n\n\n\tPlease Press Enter...");
```

```
getch();
```

```
}
```

14. Write a C program to find sum of n elements entered by user. To perform this program, allocate memory dynamically using calloc() function.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int n,i,*ptr,sum=0;
    printf("Enter number of elements: ");
    scanf("%d",&n);
    ptr=(int*)calloc(n,sizeof(int));
    if(ptr==NULL)
    {
        printf("Error! memory not allocated.");
        exit(0);
    }
    printf("Enter elements of array: ");
    for(i=0;i<n;++i)
    {
        scanf("%d",ptr+i);
        sum+=*(ptr+i);
    }
    printf("Sum=%d",sum);
    free(ptr);
    return 0;
}
```

15. Write a C program to swap numbers in cyclic order using call by reference.

```
#include<stdio.h>
void Cycle(int *a,int *b,int *c);
int main(){
    int a,b,c;
    printf("Enter value of a, b and c respectively: ");
    scanf("%d%d%d",&a,&b,&c);
    printf("Value before swaping:\n");
    printf("a=%d\nb=%d\nc=%d\n",a,b,c);
    Cycle(&a,&b,&c);
    printf("Value after swaping numbers in cycle:\n");
    printf("a=%d\nb=%d\nc=%d\n",a,b,c);
    return 0;
}
```

```

}

void Cycle(int *a,int *b,int *c){
    int temp;
    temp=*b;
    *b=*a;
    *a=*c;
    *c=temp;
}

```

16. Write a C program to add two distances entered by user in feet-inch system. To perform this program, create a structure containing elements feet and inch. [Note: 12 inch= 1feet]

```

#include <stdio.h>
struct Distance
{
    int feet;
    float inch;
}d1,d2,sum;
int main()
{
    printf("1st distance\n");
    printf("Enter feet: ");
    scanf("%d",&d1.feet);
    printf("Enter inch: ");
    scanf("%f",&d1.inch);
    printf("2nd distance\n");
    printf("Enter feet: ");
    scanf("%d",&d2.feet);
    printf("Enter inch: ");
    scanf("%f",&d2.inch);
    sum.feet=d1.feet+d2.feet;
    sum.inch=d1.inch+d2.inch;
    //If inch is greater than 12, changing it to feet.
    if (sum.inch>12){
        ++sum.feet;
        sum.inch=sum.inch-12;
    }
    printf("Sum of distances=%d\%-%.1f\"",sum.feet,sum.inch);
    return 0;
}

```

SUMMARY

- Pointer variable represents the address of a data item.
- The pointer variable might be belonging to any of the data type such as int, float, char, double, short etc.
- A pointer that is assigned NULL is called a **null** pointer.
- Two pointer variables can be compared provided both points to objects of same type.
- When a pointer is incremented, its value is incremented by the length of the data type it points to.
- In an array, the name of an array stands for the address of its zeroth element. Similarly struct variable represents the address of its first element.
- Instead of using '*', '.' and parenthesis, the -> operator. (*arrow operator.*) can be used.
- A pointer pointing to a structure just the same way a pointer pointing to an int, such pointers are known as structure pointers.
- Using the pointer, the memory location can directly be accessed.
- & (address of 0 and * (indirection operator) are the two operators used in pointer related operations.
- Pointer variables can be used in expressions.
- Addition of two pointers, multiplication, and division of the pointer with a number is not allowed.
- Array elements can be accessed through the pointers quickly.
- Pointers can be passed in the function as an argument.
- Like any other return type, functions can also return the pointers.
- The process of allocating memory at run time is known as dynamic memory allocation. Dynamic memory management technique is used to optimize the use of storage space.
- malloc(), realloc(), calloc() and free are used in dynamic memory allocation.
- The name malloc stands for "memory allocation". This function is used to allocate a block of memory. The required amount of byte should be specified as argument to the function. After allocating memory in the heap (free memory) the function returns the starting address of the block of memory allotted.
- *The storage space allocated dynamically has no name and therefore its contents can be accessed only through a pointer.*
- free() is used to deallocate the memory. The release of storage space is important when the storage is limited. When data stored in a block of memory is not needed, then release that portion of the memory.
- *by default the memory allocated by malloc() contains garbage values whereas that allocated by calloc() contains all zeros.*
- *While malloc allocates a single block of storage space, calloc allocates multiple block of storage, each of the same size, and then sets all bytes to zero.*
- The name calloc stands for "contiguous allocation" calloc function is used when requesting memory space at run time for storing derived data types such as arrays and structure..
- This function is used to change the memory size previously allocated. The space may be increased or decreased.

REVIEW QUESTIONS AND PROGRAMS

PART - A (2 Marks Questions)

1. What are pointers?
2. Explain the use of (*) indirection operator.
3. How to declare pointer variable? Give an example.
4. What are the two operators associated with the pointer variable?
5. Why is the addition of two pointers impossible?
6. Which arithmetic operations are possible with pointers?
7. How is a pointer initialized?
8. What is the result of adding an integer to a pointer.
9. What is the purpose of realloc() function?
10. What is the need for dynamic memory allocation?
11. What are the functions used in dynamic memory allocation?
12. What is a scale factor? Write down the scale factor of int, float and char data types?

PART - B (3 Marks Questions)

1. Write down the advantages of pointers?
2. What is NULL pointer? Is it the same as an uninitialized pointer?
3. What is an array of pointer? How is it declared?
4. Distinguish between (*m) [5] and *m [5]
5. Differentiate between p and *p.
6. Differentiate between calloc(0 and malloc(0 function in C.
7. How to pass the whole array to the function?
8. Why we have to release the memory and how?
9. What are the advantages and limitations of storing strings in an array of pointers ?

PART - C (5 or 10 Marks Questions)

1. What are pointers? Why are they needed? Explain with an example.
2. Explain the relation between array and pointer. Give an example.
3. Write a program using pointers to read in an array of integers and print its elements in reverse order.

4. Briefly explain about pointer arithmetic.
5. Explain the role of pointers in strings with examples.
6. How do we use array of pointers/ Explain.
7. Explain call by reference with a C program.
8. Write a program to reverse the characters of a string using pointers.
9. Write a C program to find the length of the string using a pointer.
10. Write a C program to find the average of elements of an array using pointers.
11. What do you understand by a pointer to a pointer/ Can this be extended to any level? Explain.
12. Explain the various functions used in dynamic memory management.
13. Explain the use of pointers in handling arrays and structures.
14. How can array of string pointers be initialized? Illustrate your answer with an example,

UNIT - V

FILE MANAGEMENT AND PREPROCESSORS

OBJECTIVES

At the end of the unit, the students will be able to

- Define the term “File”
- Understand the functions used in opening, closing , reading and writing a file.
- Learn the Input/output operations on files.
- Get acquaintance with the error handling during I/O operations.
- Differentiate Random access with sequence access of files.
- Learn the syntax and usage of functions used in random access.
- Write programs using files.
- Introduce command line arguments.
- Develop programs using command line arguments.
- Understand preprocessor.
- Learn about the different types of pre processor directives.

INTRODUCTION

When the program is terminated, the entire data is lost in C programming. For permanent storage of data, it is required to use files and read it from them. C language supports number of functions that have the ability to perform the basic file operations. C Pre-processor is a collection of special statements, called, directives, that are executed at the beginning of the compilation process.

This unit will discuss about the various operations that can be performed on files, concept of command line arguments and preprocessor

5.1. FILE MANAGEMENT

5.1.1. INTRODUCTION

Files are used to store a large amount of data in a disk. Handling of large volume of data through keyboards has the following disadvantages:

1. It is time consuming.
2. The entire data is lost when either the program is terminated or when the power is turned off.

A file is a group of related data, stored on the disk.

5.1.2. DEFINING AND OPENING A FILE

Before using any files, it must be opened properly. Opening a file establishes a link between the program and the operating system, about, which file we are going to access and how. The link between our program and the operating system is structure called FILE, which has been defined, in the header file "stdio.h" (standard Input Output header file). Also the purpose of opening the above file should also be defined. The purpose may be either reading, writing or appending data.

The function fopen() is used for opening a file. The general format for defining and opening a file is as follows.

```
FILE *ptr;  
ptr = fopen("filename", "mode");
```

where FILE is a defined data type. Data structure of a file is defined as FILE. It is not necessary to define this structure. The FILE is a type of structure, which has been defined in the header file stdio.h. Each file will have its own FILE structure. The FILE structure contains information about the file being used, such as its current size, its location in memory etc . So all the files should be declared as type FILE before they are used. This structure is defined only in uppercase letters. Do not use file or File instead of FILE.

ptr is a pointer variable containing the address of the structure FILE. ptr is a pointer to the data type FILE.

The second statement opens the file indicated as filename and assigns an identifier to the FILE type pointer ptr. "mode" part is used to indicate the purpose for which the file is being opened. The value of "mode" can be any one of the following.

Mode	Purpose	Action
"r"	Read from the file	This mode searches the disk for the filename. If it exists, then it is loaded from disk into the memory and a pointer returned to the file; otherwise, an error occurs and returns NULL value. NULL indicates that there will be a failure in opening the file.
"w"	Write data into the file.	A file with a specified name is created, if the file does not exist. If the file already exists, its contents are deleted. If it is unable to open the file, it returns NULL.
"a"	Append data to the file.	If the file already exists, it is opened so that data may be added. If the file does not exist, it is created. If it is unable to open the file, it returns NULL.
r+	-	Opens a text file for reading and writing both
w+	-	Opens a text file for reading and writing both. It first truncates the file to zero length if it exists otherwise creates the file if it does not exist.

a+	-	Opens a text file for reading and writing both. It creates the file if it does not exist. The reading will start from the beginning but writing can only be appended.
----	---	---

Trouble in file opening :

The fopen() function for opening file for read/write operations may fail due to anyone of the following reasons:

- A file for reading may not be present on the disk
- Insufficient disk space for opening a for writing.
- Write protected disk does not allow storage of data on it.
- Dealing with a corrupt file

Example

```
FILE *ptrf;
ptrf = fopen ("exam.dat", "r");
```

The above two statements are used to open the file “exam.dat” for reading and the pointer ptrf returning a pointer. If the file “exam.dat” does not exist, an error will occur and the value NULL is assigned to ptrf.

The following program is used to display the error message if the file could not be opened due to some reasons.

```
#include <stdio.h>

main()
{
    FILE *ptrf;
    ptrf = fopen("exam.dat","r");
    if ( ptrf == NULL)
    {
        printf( " File could not be opened \n");
        exit (0);
    }
    -----
}
```

A number of files can be opened at a time.

Difference between Append and Write Mode :

Write (w) mode and Append (a) mode, while opening a file are almost the same. Both are used to write in a file. In both the modes, new file is created if it doesn't exists already. The only difference they have is, when opening a file in the write mode, the file is reset, resulting in deletion of any data already present in the file. While in append mode this will not happen. Append mode is used to append or add data to the existing data of file (if any). Hence, when opening a file in Append(a) mode, the cursor is positioned at the end of the present data in the file.

5.1.3. CLOSING A FILE

A file must be closed immediately after finishing all the operations on the above file. The purposes of closing a file is as follows:

1. To flush the information from the buffers of the memory.
2. To break all the links to the file.
3. To prevent the accidental movement of the file.
4. To reopen the same file in a different mode.

The general format for closing a file is

```
fclose (ptrf) ;
```

The above statement close the file associated with the FILE pointer (ptrf). Once a file is closed, its file pointer can be reused for another file.

Example

```
FILE ptrf1, ptrf2;  
ptrf1 = fopen("exam.dat", "r");  
ptrf1 = fopen("result.dat", "w");  
-----  
fclose (ptrf1);  
fclose (ptrf2);
```

The above program opens two files and closes them after all operations are completed.

5.1.4. INPUT/OUTPUT OPERATIONS ON FILES

Reading from a file (getc() function)

Once the file has been opened for reading using `fopen()`, the file's contents are brought into memory (partly or wholly) and a pointer points to the very first character. To read the file's contents from memory there exists a standard library function called `getc()`.

`getc()` function is used to read a character at a time from the file. When a file is opened, the file pointer provides access to the first character written on the file.

The functions `getc()` reads one character from current pointer position,. This character is then assigned to variable and advances the pointer position so that it points to the next character. Once the file has been opened, it is referred by file pointer and file name is no longer valid.

`getc()` function stops the reading process when the end of file is reached. The end of file can be checked by checking for EOF.

For example, the statement

```
ch = getc(fp1);
```

read a character from the file whose file pointer is `fp1`. The above character is stored in the variable `ch`. `getc()` and `fgetc()` are both same.

Program 1 : Write a program to read file and print the contents on the screen.

```
#include<stdio.h>
main()
{
    FILE*fp; char ch;
    char filename[12];
    printf("\n\nEnter the filename to read:");
    scanf("%s",filename);
    fp=fopen(filename,"r");
    do
    {
        ch=fgetc(fp);
        printf("%c",ch);
    } while(ch!=EOF);
    fclose(fp);
    getch(); }
```

Program 2 : Read a file called “bio.dat” and count the number of characters in it.

```
#include<stdio.h>
main()
{
    FILE*fp;
    int count=0;
    char ch;
    fp=fopen("bio.txt","r");
    do
    {
        ch=fgetc(fp);
        count++;
    } while(ch!=EOF);
    printf("\n\nThe no. of characters in a file is:%d",count);
    printf("\n including EOF character");
    fclose(fp);
    getch(); }
```

Writing into a file (putc() function)

putc() function is used to write a character into a file. The statement

```
putc( c, ptrf);
```

write the character contained in the character variable c to the file associated with FILE pointer ptrf.

The end of file is marked by EOF. This EOF can be placed at end of file through ^ z(ctrl + z). putc() and fputc() are same.

Program 3 : Create a text file called “self.dat”.

```
#include<stdio.h>
main() {
FILE*fp;
char ch;
clrscr();
fp=fopen("self.txt","w");
do {
    ch=getche();
    putc(ch,fp);
} while(ch!=13);
fclose(fp); }
```

The getw() and putw() function

The getw() and putw() are integer oriented functions. These two functions are used to read and write integer values. These functions are useful when handling only integer data. The general format of getw and putw function are

getw(ptrf);	putw(integer, ptrf);
-------------	----------------------

Program 4 : Create a file “data.dat” which contain 10 integer numbers. From the above file create another file, which contains only odd numbers, and print the contents of this file.

```
#include<stdio.h>
main(){
FILE *fp1,*fp2;
int num,i;
clrscr();
fp1=fopen("data.txt","w");
for(i=0;i<10;i++)
{
    scanf("%d",&num);
    putw(num,fp1);}
fclose(fp1);
fp1=fopen("data.txt","r");
fp2=fopen("odd.txt","w");
while((num=getw(fp1))!=EOF)
if((num%2)!=0)putw(num,fp2);
fclose(fp1);
fclose(fp2);
fp1=fopen("data.txt","r");
clrscr();
printf("All Numbers - data.dat\n\n");
    while((num=getw(fp1))!=EOF)
printf("%d",num);
fp2=fopen("odd.txt","r");
printf("\n\nOdd Numbers-odd.dat\n\n");
while((num=getw(fp2))!=EOF)
printf("%d",num);
fclose(fp1); fclose(fp2); }
```

5.1.5. FORMATTED FUNCTIONS (`fprintf()` AND `fscanf()`)

`getc()`, `putc()`, `getw()` and `putw()` function can handle one character or integer at a time. But the functions, `fprintf()` and `fscanf()` can handle a group of data simultaneously.

These two functions perform input/output operations that are identical to `printf` and `scanf` functions. But these two functions work on files.

The parameters to these functions are similar to `printf` and `scanf`. But one more additional parameter, FILE pointer specifying the file, is added to these functions. The first argument of these functions is a file pointer, which specifies the file to be used.

fprintf()

The general format of `fprintf` statement is

```
fprintf( ptrf, "control_string", list);
```

where `ptrf` is the file pointer for the file that has been opened for writing. The control string specifies the output format for the various items in the list.

The list may contain variables, constants or strings.

Examples

```
fprintf( ptrf1, "%s %d %d ", name , age , salary);
```

```
fprintf( ptrf1, "%2f %d %d ", price , quantity, product_name);
```

fscanf()

`fscanf` reads characters from the specified file, converts them as per directions given in control string format. Then the converted value is assigned to the objects pointed by the list of arguments. The general form is

```
fscanf( ptrf, "control_string", list);
```

Examples

```
fscanf( ptr3, "%s %d %d ", &name , &age , &salary);
```

```
fscanf( ptrf1, "%f %d %d ", &price , &quantity, &product_name);
```

The function returns the value EOF if an input failure occurs.

Program 5: Write a c program to get inputs as name, roll no, age and native place from the keyboard and store the details in a data file. Also read the file and display formatted.

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

main()
```

```

{
FILE*fp;
char filename[12];
char name[30],rollno[5],nat_place[20],ch;
int age,i,end;
clrscr();
printf("Enter the filename to read and write:");
scanf("%s",filename);
fp=fopen(filename,"w");
do
{
    clrscr();
    printf("Enter the name,rollno,age and native place::\n");
    fscanf(stdin,"%s%s%d%s",name,rollno,&age,nat_place);
    fprintf(fp,"%s%s%d%s",name,rollno,&age,nat_place);
    fflush(stdin);
    printf("\n\nDo U want to continue?");
    ch=getch();
} while((ch=='y')||(ch=='Y'));
end = ftell(fp);
fclose(fp);
fp=fopen(filename,"r");
printf("\n\nName\tRollno\tAge\tNat_Place\n\n");
while(ftell(fp)<end)
{
    fscanf(fp,"%s%s%d%s",name,rollno,age,nat_place);
    fprintf(stdout,"%-30s\t%-5s\t%5d\t%-20s\n",name,rollno,age,nat_place);
}
fclose(fp);
}

```

5.1.6. ERROR HANDLING DURING I/O OPERATIONS

An error may occur during input / output operations on a file. Some of the situations in which the error occurs are as follows.

1. When reading the data beyond the EOF mark.
2. Device overflow (i.e., no space in disk)
3. Not opening of files.
4. Trying to perform an operation on a file, when the file is opened for another type of operation. For example it is not possible to perform read operation, when the file is opened for write operation.
5. Opening a file with an invalid file name.

Two library functions *feof* and *ferror* is used to detect I/O errors in the files.

feof()

This function is used to test for an end of file condition. This function has only one argument. This argument is a FILE pointer. This function returns a non-zero integer value if all the data from the specified file has been read. Otherwise it returns zero. For example, the statements

```

if(feof(ptrf));
    printf( " End of file");

```

displays the message “End of file”, on reaching the end of file condition.

ferror()

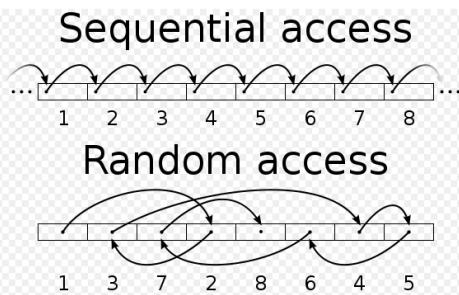
This function is used to report the status of the file specified. This function also takes only one argument, FILE pointer. This function returns a non-zero integer if an error has been detected upto that point during processing. Otherwise, it returns zero. For example, the statements

```
if(ferror(ptrf)!=0);  
printf (" There will be an error");
```

display the message “There will be error”, if the reading is not successful.

5.1.7. RANDOM ACCESS FILES

Sequential and Random Access File Handling in C



In computer programming, the two main types of file handling are:

- Sequential;
- Random access.

Sequential files are generally used in cases where the program processes the data in a sequential fashion – i.e. counting words in a text file – although in some cases, random access can be feigned by moving backwards and forwards over a sequential file.

True random access file handling, however, only accesses the file at the point at which the data should be read or written, rather than having to process it sequentially.

In random access mode, it is possible to access only a particular part of a file. The functions available for random access operations are (i)fseek (ii) ftell and (iii) rewind.

To access the forty fourth record then first forty three record read sequentially to reach forty four record . In random access data can be accessed and processed directly .There is no need to read each record sequentially . To access a particular record in a random access takes less time than the sequential access

The ftell() function

This function returns the current offset position in the file. For example, as soon as a file is opened, this function returns 0. this function is used to save the current position of a file, which can be used later in the program. The general format is

```
long int position = ftell(ptrf);
```

where position would give the relative offset (in bytes) of the current position.

The rewind() function

This function resets the pointer position to the beginning of the file. For example, the statements

```
rewind(ptrf);  
position = ftell(ptrf);
```

assigns the value 0 to position. The first byte in the file is numbered as 0, second as 1 and so on.

This function is used to read a file more than once, without having to close and open the file.

fseek () function

After performing an input or output operation on a file, the pointer is shifted to the next position in the file. The “current position” is the position at which data will be read from in an input operation or written to in an output operation. When a file is opened, the “current position”, is always at the beginning of the file.

fseek function is used to move the file position to a desired location. The general format of fseek function is

```
fseek(fileptr, offset,  
from_whre);
```

where *file-ptr* is a pointer to the file concerned. *Offset* is a long integer that specifies the number of bytes that have to be shifted. *from_ where* is an integer specifying the position on the file from where the offset would be effective. *from_ where* can take one of the following three values.

Value	Meaning
0	Offset is effective from beginning of file
1	Offset is effective from the current position
2	Offset is effective from end of file

The offset may be positive or negative. If it is positive, the pointer moves forwards. If it is negative the pointer moves backwards.

Examples

```
fseek(fp,0L,0) - beginning of the file ( similar to rewind)  
fseek(fp,0L,1) - stay at current position  
fseek(fp,0L,2) - Go to the end of the file  
fseek(fp,m,0) - Move to (m+1) th byte in the file.
```

when the operation is successful, fseek returns 0; otherwise it returns 1.

Example

```
fp = fopen ("mark.dat", "r");
```

```
fseek ( fp,0,0);
```

0	1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---	---	-------	-------



fseek (fp,5,0)

0	1	2	3	4	5	6	7	8	9
↑											

fseek(fp, -2,1)

0	1	2	3	4	5	6	7	8	9
↑											

Current position shifted by – 2 bytes.

Program 6 : Write a program that displays the content of a file in reverse order.

```
#include<conio.h>
#include<stdio.h>
main()
{
    FILE *fp;
    char ch;long int pos;
    clrscr();
    fp=fopen("s","r");
    pos=f.tell(fp);
    fseek(fp,-1,2);
    while(f.tell(fp)>pos)
    {
        ch=fgetc(fp);
        printf("%c",ch);
        fseek(fp,-2,1);
    }
    ch=fgetc(fp);
    putchar(ch);
    fclose(fp);
    getch(); }
```

In the above program, the statement *(fp, -1,2)* is used to move the file pointer to the last character.

Since every read/write operation moves the file pointer forward by one position, the file pointer is moved back by two positions to read a next character. This is achieved by the statement *feek(fp-2,1)*.

FILE PROGRAMS

Write a C program to find a read a text file and copy it to other file such that each word comes in separate lines.

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

```

{
FILE *fp1,*fp2;
char infile[12],outfile[12],ch;
clrscr();
printf("Enter the file name to read:");
scanf("%s",infile);
printf("Enter the file name to write ");
scanf("%s",outfile);
fp1=fopen(infile,"r");
fp2=fopen(outfile,"w");
do{
ch=fgetc(fp1);
if(ch==' ')
fputc('\n',fp2);
else
fputc(ch,fp2);
}
while(ch!=EOF);
fclose(fp1);
fclose(fp2);
}

```

Write a program to convert lowercase text file to uppercase text file.

```

#include<conio.h>
#include<stdio.h>
main()
{
FILE *fp1,*fp2;
char lowerfile[12],upperfile[12],ch;
clrscr();
printf("Enter the file name to read:");
scanf("%s",lowerfile);
printf("\nEnter the file name to write :");
scanf("%s",upperfile);
fp1=fopen(lowerfile,"r");
fp2=fopen(upperfile,"w");
do{
ch=fgetc(fp1);
if((ch>='a')&&(ch<='z'))fputc(ch-32,fp2);
else
fputc(ch,fp2);
}while(ch!=EOF);
fclose(fp1);
fclose(fp2);
getch();
}

```

Write a Program to remove all the comment lines in a C program.

```
#include<conio.h>
#include<stdio.h>
void main(int argc ,char *argv[])
{
    FILE *fp,*tmp;
    char comment[3],ch;
    int flag=1,f1,i,pt=0;
    int start ,end;
    fp=fopen(argv[1],"r");
    tmp=fopen("temp","w");
    start=fteill(fp);
    fseek(fp,-1,2);
    end=tell(fp);
    f1=end-start;

    for (i=0;i<f1;i++,pt++)
    {
        fseek(fp,pt,0);
        fgets(comment,3,fp);
        if(strcmp(comment,"/*")==0)flag=0;
        else if(strcmp(comment,"*/")==0)
        {
            flag=1;
            pt=fteill(fp)+2; /*for avoid \n character*/
        }
        if( flag)
        {
            fseek(fp,pt,0);
            ch=fgetc(fp);
            if(ch=='\n')
                pt++;
            fputc(ch,tmp);
        }
    }
    fclose(fp);
    fclose(tmp);
}
```

Write a C program to read from the keyboard and write the following data pertaining to the supply of stores in a binary file using function and write();

Supplier	Address	Part No	Quantity
Ram Bros	12, Anna Salai	2345	45
SPK	345, Patel Road	567	78

Use a structure for the fields in each record. Write a function to display the data on the screen

```
#include<stdio.h>
#include<conio.h>
struct supplier
{
char name[20];
char addr[30];
int partno;
int qty;
}supp;
main()
{
FILE *fp;
char filename[12],ch;
int i,end;
clrscr();
printf("Enter the file name to write and read:");
scanf("%s",filename);
fp=fopen(filename,"w");
do
{
clrscr();
printf("Enter the supplier name ,address,partno,quantity:\n");
fscanf(stdin,"%s %s %d %d ",supp.name,supp.addr,&supp.partno,&supp.qty);
fprintf(fp,"%s %s %d %d\n",supp.name,supp.addr,supp.partno,supp.qty);
fflush(stdin);
printf("\n\n Do you want to continue?");
ch=getche();
}while((ch=='y')||(ch=='Y'));
end =ftell(fp);
fclose(fp);
fp=fopen(filename,"r");
printf("\n\n Supplier\tAddress \tPartno\tQuantity\n\n");
while(ftell(fp)<end)
{
fscanf(fp,"%s%s%d%d",supp.name,supp.addr,&supp.partno,&supp.qty);
fprintf(stdout,"%-20s\t%-30s%5d%5d\n",supp.name,supp.addr,&supp.partno,&supp.qty);
}
fclose(fp);
getch(); }
```

5.2. COMMAND LINE ARGUMENTS

It is possible to pass some values from the command line to C programs when they are executed. These values are called **command line arguments** and many times they are important to control the program. Command line arguments are parameters. These parameters are supplied to a program when the program is executed. The first parameters may represent a file name the program should process.

main() function can also take arguments like other functions. main() can take two arguments. So when main is called, it is called with two arguments. They are *argc* and *argv*. The information contained in the command lines is passed onto the program through these arguments.

The *argc* is called as an “argument counter” and it represents the number of command line arguments. The *argv* is an argument vector, is a pointer to an arrays of character strings that containing the arguments, one per string. The size of this array will be equal to the value of *argc*.

The c in *argc* stands for counter and v in *argv* stands for vector.

In order to access the command line arguments, main function is declared as follows:

```
main (int argc, char *argv[])
{
-----
}
```

The first parameter in the command line is always the program name. Therefore *argv[0]* represents the program name.

Example

Assume that there is a program called “display” which displays whatever input string is fed to it. The following program is invoked with the command line argument “*display Good morning to everybody*”. Then the following program displays the message “Good morning to everybody”.

```
main (int argc, char *argv[])
{
    int i = 1;
    while ( -- argc > 0)
    {
        printf ( "%s", argv[i]);
        if ( i < argc - 1)
            printf( "\n");
        else
            printf(" ");
    }
}
```

OUTPUT:

C:\TC\BIN > *display Good morning to everybody*

Good morning to everybody

When the command is processed, the value of *argc* is 5.

Argv	Array of strings	
.	0	→ Display\0
	1	→ Good\0
	2	→ morning\0
	3	→ to\0
	4	→ everybody\0
	5 NULL	

argv[0] is always the name of the program and argv[argc] is NULL.

C Program to Add two numbers using Command Line Arguments

```
#include<stdio.h>
void main(int argc, char * argv[])
{
    int i, sum = 0;
    if (argc != 3) {
        printf("You have forgot to type numbers.");
        exit(1);
    }
    printf("The sum is : ");
    sum = sum + atoi(argv[1]) + atoi(argv[2]);
    printf("%d", sum);
}
```

Write a Simple Calculator program using command line arguments

```
#include<stdio.h>
#include<conio.h>
int main(int argc, char *argv[]) {
    float result = atoi(argv[1]);
    int i;
    for (i = 3; i < argc; i = i + 2)
        switch (*argv[i-1]) {
            case '+':
                result = result + atoi(argv[i]);
                break;
            case '-':
                result = result - atoi(argv[i]);
                break;
            case '*':
                result = result * atoi(argv[i]);
                break;
            case '/':
```

```

result = result / atoi(argv[i]);
break;
default:
printf("Wrong Input");
}
printf("\nResult = %f", result);
}

```

Write a program to copy the contents of one file to another file (Read file names using command line arguments).

```

#include <stdio.h>
main (int argc, char *argv[]);
{
    FILE *fp1, *fp2;
    char ch;
    fp1 = fopen(argv[1], "r");
    fp1 = fopen(argv[2], "w");
    do
    {
        ch = fgetc(fp1);
        fputc(ch, fp2);
    } while(ch != EOF);
    printf ("File copied"

    fclose(fp1);
    fclose(fp2);
}

```

5.3. PREPROCESSOR

5.3.1. INTRODUCTION

Preprocessor processes the source code before the compilation begins. The preprocessor is invoked automatically by the C compiler.

In C program, commands or instructions to the C compiler can be included. These are called preprocessor directives. Every preprocessor directive must begin with the character #. Preprocessor directive do not require a semicolon at the end. Preprocessor directives are placed in the source program before the main function.

Preprocessor directives can be subdivided into

- Macro definition and substitution
- File inclusion.
- Compiler controlled directives.

5.3.2. MACRO DEFINITION DIRECTIVES

In macro substitution, an identifier is replaced by a string. The preprocessor accomplishes this task with the help of define statement. The general form of define statement is

```
# define macro_name macro body
```

The above statement is used to replace the macro_name wherever it occurs in the program by the string of the macro_body. Macro_name is generally written in **uppercase letters**.

The **#define** directive defines an identifier and a string/constant that is substituted in place of the identifier name each time it is encountered in the file. This identifier is called **macroname** and the replacement process is known as **macro substitution**.

There is no need of a semicolon after the statement having a #define directive.

Advantages of #define statement :

1. It helps in generation of faster and compact code.
2. It saves the programmer's time.
3. It reduces the chances of inconsistency within the program.
4. It makes the modification easier, as the value has to be changed only at one place in the program i.e., it increases the flexibility of the program

Macro Substitutions

Simple macro substitution is commonly used to define constants

Examples

```
# define PI 3.14  
# define PLACE "SALEM"  
# define SUM 0
```

Expressions can also be included in macro definition

For example,

```
# define AREA 10 * 25.5  
# define THREE_PI 3 * 3.1
```

are also valid . Following examples are also the correct macro definitions.

```
# define EQUALS ==  
# define AND &&  
# define OR ||  
# define NOT_EQUAL !=  
# define INCREMENT ++
```

Then the following statement

```
if (age EQUALS 25 AND height NOT_EQUAL 50) INCREMENT count; is also valid.
```

Macro with arguments

Macros with arguments takes the following form

```
# define variable name(a1,a2.....an), string
```

There should not be any space between variable name and the left parentheses. The arguments a1, a2, are formal macro arguments.

Example

```
#define square (A) (A*A)
```

if the following statement appears late in the program

```
AREA = square(side);
```

Then the above statement is equivalent to : AREA = side * side;

Example

```
# define MAX (X,Y)((X) >(Y))?(X:Y)
# define MIN(A,B)((A) <(B)) ? (A):(B)
# define ABS(X) (((X)>0) ? (X):(-X))
```

Example

```
#define AREA(x) ( 3.14 * x* x )
main( )
{
float r1 = 6.25, r2 = 2.5 , a;
a = AREA ( r1 ) ;
printf ( "Area of circle = %f\n", a ) ;
a = AREA ( r2 ) ;
printf ( "Area of circle = %f", a ) ;
getch();
}
```

Difference between functions and macros in C

- Macro templates are replaced by macro expansion by preprocessor. Whenever function is called, execution goes to that function, calculations are performed and execution returns to main.
- Macros are faster than functions.
- In macros program size increases if called for more times.
- In function program remains compact even though called for several times.
- No address is associated with macro identifier and hence pointers cannot be used with macros.
- When we pass the arguments to the macro, it never checks for their data types and checks for only the number of arguments

Undefining a Macro

undef directive is used to undefine a defined directive. The statement

```
# undef macro_name
```

cause the previous preprocessor directive with this macro_name to lapse. From this point onwards, the previous definition will not hold good. undef is useful when restricting the definition only to a particular part of the program,

Major advantage of using macro is to increase the speed of the execution of the program.

Disadvantages of Macros

- No type checking is performed in macro. This may cause error,
- A macro call may cause unexpected results.

5.3.3. FILE INCLUSION

External files containing function or macro definitions can be inserted into the program by the preprocessor directive *include*. The general format for include directive is

```
# include "filename", (or) # include <filename>
```

Where *filename* is the name of the file to be inserted. The preprocessor inserts the entire contents into the source code of the program.

When the *file name* is within the double quotation marks, the file is first searched in the current directory and then in the standard directories. When the *file name* is within a pair of angle brackets, the file is searched only in the standard directories.

5.3.4. CONDITIONAL COMPIRATION

Compiling the selected portion of the program for a particular condition is called “conditional compilation”. Conditional compilation directives are (i) ifdef (ii) ifndef (iii) #if #else #endif directive **ifdef**

The #ifdef feature is used to remove sections of codes automatically. For example,

```
# ifdef LABEL
{
    Block _1
}
#else
{
    Block _2
}
#endif
```

Block 1 will get compiled only if LABEL has been # defined. If LABEL has not been defined as a macro, the Block1 won't be sent for compilation at all. Block 2 will get compiled.

For example, assume a single program can be compiled by two different makes of computer. The following single program conditionally compiles only the code pertaining to either of the two machines.

```
main()
{
    # ifdef MACINTOSH
    {
        code for MACINTOSH
    }
    # else
    {
        code for IBM
    }
    # endif
}
```

#ifdef means “if defined”. The above program would run smoothly on a MACINTOSH as well as IBM. If MACINTOSH has been defined, then the above program will run on MACINTOSH computer. Otherwise it will run on IBM computer. For running the above program in Macintosh, the directive # define MACINTOSH may be included.

Example

```
# define SQR
main()
{
    int i = 3;
    # ifdef SQR
    {
        printf ("Square of I = %d", I*I);
    }
    # else
    {
        printf (" i = %d", i);
    }
    # endif
}
```

The output of the above program is 9.

ifndef directive

ifndef means “if not defined”. This directive works exactly opposite to the directive #ifdef. The general form of this directive is

```
#ifndef macro_name
statement 1;
```

```
#else
    statement2;
#endif
```

If there is no #define for macro_name, then statement1 will be executed otherwise statement2.

#if #else #endif directive

#if #else #endif directive tests an expression for non zero value and performs the conditional compilation, The general form of this is:

```
# if constant-expression
    Statement;
#else
    Statement2;
```

PROGRAMS USING MACROS

Write a Macro to determine the absolute value of a numeric data and use the macro in your program.

```
#include<stdio.h>
#include<math.h>
#define absolute(x) abs(x)
main()
{
    int num;
    clrscr();
    printf("\n Enter the number:");
    scanf("%d",&num);
    printf("\n The absolute value is:%d",absolute(num));
}
```

Write a nested macro that gives the minimum of three values.

```
#include<stdio.h>
#include<conio.h>
#define min(a,b) ((a>b)?b:a)
#define minthree(a,b,c) (min(min(a,b),c))
void main()
{
    int x,y,z,w;
    clrscr();
    printf("enter three numbers :\n");
    scanf("%d%d%d",&x,&y,&w);
    z=minthree(x,y,w);
    printf("Minimum of three value is %d",z);
    getch();}
```

ADDITIONAL PROGRAMS

Write a program which accepts two file names from the command line and prints whether the contents of the two files are same or not. If not, then print the first line number and then the contents of the lines from which they differ. Assume that the lines in both the files have at most 80 characters.

```
#include<stdio.h>
main(int argc, char *argv[])
{
FILE *fp1, *fp2;
char ch1[80],ch2[80];
int i=1,j=1,l1=0,l2=0;
if (argc!=3)
{
printf ("\nWrong number of arguments.");
getch();
exit();
}
fp1=fopen(argv[1],"r");
if (fp1==NULL)
{
printf ("\nunable to open the file %s",argv[1]);
getch();
exit();
}
fp2=fopen(argv[2],"r");
if (fp2==NULL)
{
printf ("\nunable to open the file %s",argv[2]);
getch();
exit();
}
l1=0;
l2=0;
while (i!=0 && j!=0)
{
i=fgets(ch1,80,fp1);l1++;
j=fgets(ch2,80,fp2);l2++;
if (strcmp(ch1,ch2)!=0)
{
printf ("\nContents of both Files are not
Equal");
printf ("\nLine Number\tContents\tFile

```

```

name\n");
printf ("%d\t\t%s\t\tFrom %s",l1,ch1,argv[1]);
printf ("%d\t\t%s\t\tFrom %s",l2,ch2,argv[2]);
exit();
}
if (i==0 && j==0)
printf ("\nBoth Files are equal");
else
printf("\nBoth files are not equal");
getch();
}

```

Write a C program to create a file contains a series of integer numbers and then reads all numbers of this file and write all odd numbers to other file called odd and write all even numbers to a file called even.

```

#include<stdio.h>
#include<conio.h>
void main()
{
FILE *f1,*f2,*f3;
int i,j;
printf("Enter the data\n");
f1=fopen("file1","w");
for(i=0;i<=10;i++)
{ scanf("%d",&j);
if(j== -1) break;
putw(j,f1);
}
fclose(f1);
f1=fopen("file1","r");
f2=fopen("od","w");
f3=fopen("ev","w");
while((j=getw(f1))!=EOF)
{ if(j%2==0)
putw(j,f3);
else
putw(j,f2);
}
fclose(f1);
fclose(f2);
fclose(f3);
f2=fopen("od","r");
f3=fopen("ev","r");
printf("\nContents of odd file\n");

```

```

while((j=getw(f2))!=EOF)
printf("%4d",j);
printf("\nContents of even file\n");
while((j=getw(f3))!=EOF)
printf("%4d",j);
fclose(f2);
fclose(f3);
getch();
}

```

Write a C program to read name and marks of n number of students from user and store them in a file.

```

#include <stdio.h>
#include<conio.h>
int main(){
    char name[50];
    int marks,i,n;
    clrscr();
    printf("Enter number of students: ");
    scanf("%d",&n);
    FILE *fptr;
    fptr=(fopen("C:\\student.txt","w"));
    if(fptr==NULL){
        printf("Error!");
        exit(1);
    }
    for(i=0;i<n;++i)
    {
        printf("For student%d\\nEnter name: ",i+1);
        scanf("%s",name);
        printf("Enter marks: ");
        scanf("%d",&marks);
        fprintf(fptr,"\\nName: %s \\nMarks=%d \\n",name,marks);
    }
    fclose(fptr);
    getch();
    return 0;
}

```

Write a C program to read name and marks of n number of students from user and store them in a file. If the file previously exists, add the information of n students.

```

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

```

```
{  
    char name[50];  
    int marks,i,n;  
    clrscr();  
    printf("Enter number of students: ");  
    scanf("%d",&n);  
    FILE *fptr;  
    fptr=(fopen("C:\\student.txt","a"));  
    if(fptr==NULL){  
        printf("Error!");  
        exit(1);  
    }  
    for(i=0;i<n;++i)  
    {  
        printf("For student%d\\nEnter name: ",i+1);  
        scanf("%s",name);  
        printf("Enter marks: ");  
        scanf("%d",&marks);  
        fprintf(fptr,"\\nName: %s \\nMarks=%d \\n",name,marks);  
    }  
    fclose(fptr);  
    getch();  
    return 0;  
}
```

SUMMARY

- A file a place on the disk where a group of related data is stored.
- For reading and writing on files, `getc()`, `putc()`, `getw()`, `puts()` functions are used.
- Opening a file establishes a link between the program and the operating system, about, which file we are going to access and how. The link between our program and the operating system is structure called `FILE`, which has been defined, in the header file "stdio.h" (standard Input Output header file).
- "r"s mode searches the disk for the filename. If it exists, then it is loaded from disk into the memory and a pointer returned to the file; otherwise, an error occurs and returns NULL value. NULL indicates that there will be a failure in opening the file.
- When opening a file in the write mode, the file is reset, resulting in deletion of any data already present in the file. While in append mode this will not happen. Append mode is used to append or add data to the existing data of file (if any). Hence, when opening a file in Append(a) mode, the cursor is positioned at the end of the present data in the file.
- `getc()` function is used to read a character at a time from the file. When a file is opened, the file pointer provides access to the first character written on the file.
- The `getw()` and `putw()` are integer oriented functions. These two functions are used to read and write integer values.
- `getc()`, `putc()`, `getw()` and `putw()` function can handle one character or integer at a time. But the functions, `fprintf()` and `fscanf()` can handle a group of data simultaneously.
- `fprintf()` and `facanf()` functions perform input/output operations that are identical to `printf` and `scanf` functions. But these two functions work on files.
- Two library functions `feof` and `ferror` is used to detect I/O errors in the files.
- In random access mode, it is possible to access only a particular part of a file. The functions available for random access operations are (i)`fseek` (ii) `ftell` and (iii) `rewind`.
- `ftell()` function returns the current offset position in the file. For example, as soon as a file is opened, this function returns 0.
- `rewind()` function resets the pointer position to the beginning of the file.
- When a file is opened, the "current position", is always at the beginning of the file. `fseek` function is used to move the file position to a desired location.
- The functions `getc()` reads one character from current pointer position.,
- For random access on files, the following functions are provided , namely , `ftell`, `rewind` and `fseek`.
- When an existing file is open using "w" mode, the contents of file are deleted.
- EOF is an integer type with a value -1. Therefore, integer variable must be used for EOF.
- It is a good practice to close all file before terminating a program.
- In the command line parameters , the first parameters may represent a file name the program should process.
- The `argc` is called as an "argument counter" and it represents the number of command line arguments. The `argv` is an argument vector, is a pointer to an arrays of character strings that containing the arguments, one per string.
- The c in `argc` stands for counter and v in `argv` stands for vector.
- Preprocessor processes the source code before the compilation begins. The preprocessor is invoked automatically by the C compiler.

- Preprocessor directives can be subdivided into Macro definition and substitution , File inclusion , Compiler controlled directives.
- The **#define** directive defines an identifier and a string/constant that is substituted in place of the identifier name each time it is encountered in the file. This identifier is called **macroname** and the replacement process is known as **macro substitution**.
- Macro templates are replaced by macro expansion by preprocessor. Whenever function is called, execution goes to that function, calculations are performed and execution returns to main.
- **Major advantage** of using macro is to increase the speed of the execution of the program.
- External files containing function or macro definitions can be inserted into the program by the preprocessor directive *include*.
- Compiling the selected portion of the program for a particular condition is called “conditional compilation”. Conditional compilation directives are (i) ifdef (ii) ifndef (iii) #if #else #endif directive.

REVIEW QUESTIONS AND PROGRAMS

PART – A (2 Marks)

1. What are the disadvantages of handling large volume of data through keyboards ?
2. Write down the general format for defining and opening a file.
3. What do you mean by file pointer?
4. What is the difference between write and append mode?
5. What do you understand by opening a data file? How is this performed?
6. Differentiate between the following : getw() and putw()
7. State the difference between getc() and fscanf() functions when handling data.
8. How to find the end of file?
9. What is the advantage of random file access methods?
10. What are the common uses of rewind and ftell functions?
11. What do the ‘c’ and “v” in argc and argv stand for?
12. What is a preprocessor?
13. How the preprocessor directives can be sub divided?
14. What is a macro? What points to be considered while defining macros.
15. What are the advantage of #define statement?
16. What are the advantage of macros?
17. What do you mean by undefining a Macro?

18. State the difference between #include "stdio.h" and #include <stdio.h>
19. What is conditional compilation?
20. Give some examples for conditional compilation directives.

PART – B (3 Marks)

1. Describe various modes of opening a data file in C
2. Mention any three purposes for closing a file.
3. What are the functions associated with random access of a file?
4. Tabulate the differences between functions and macros
5. How to use ftell() function in a file program?
6. Explain #define directive.
7. What is conditional compilation? Give an example.
8. Distinguish between #ifdef and #if directives.
9. What is the difference between the statements rewind(fp); and fseek(fp,0L,0);?

PART – C (5/10 Marks)

1. Briefly explain about the Input / Output operations on files
2. List down the situations in which errors may occur.
3. What are the two functions used to detect I/O errors in the files? Explain them.
4. Explain the syntax and usage of functions used in random access of a file with examples.
5. Write a program in C to replace every uppercase letter with corresponding lowercase letters and vice-versa in a file.
6. Write a C program to concatenate two text files.
7. Write a C program to copy a text file to another.
8. What is macro substitution? Explain with an example.
9. Explain macro with arguments with an example program
10. Define a macro CIRCUMFERENCE having argument r for finding the circumference of a circle. Write a C program to use it in main() function.
11. Define a macro INTEREST having arguments p,r, and n for finding the simple interest. Write a C program to use it in main () function to compute the simple interest for principal, rate and time as 2000, 10.0, 12.0 respectively.
12. What is file inclusion? Explain.
13. Explain about conditional directives.