GUDLAVALLERU ENGINEERING COLLEGE

(An Autonomous Institute with Permanent Affiliation to JNTUK, Kakinada)

Seshadri Rao Knowledge Village, Gudlavalleru – 521 356.

Department of Computer Science and Engineering

HANDOUT

on

PROBLEM SOLVING THROUGH COMPUTER PROGRAMMING

VISION

To be a centre of innovation by adopting changes in Information Technology, imparting quality education, research to produce visionary computer professionals and entrepreneurs.

MISSION

- To provide an academic environment in which students are given the essential resources for solving real-world problems and work in multidisciplinary teams.
- To impart value based education and research among students, particularly belonging to rural areas, for their sustained growth in technological aspects and leadership.
- To collaborate with the industry for making the students adoptable to evolving changes in Information Technology and related areas.

Program Educational Objectives

PEO1: To exhibit analytical skills in modeling and solving computing problems by applying mathematical, scientific and engineering knowledge and to pursue their higher studies.

PEO2: To communicate effectively with multi-disciplinary teams to develop quality software systems with an orientation towards research and development for lifelong

PEO3: To use emerging technologies in project development to fulfil industry and societal needs for the growth of global economy following professional ethics.

HANDOUT ON PROBLEM SOLVING THROUGH COMPUTER PROGRAMMING

Class & Sem: I B.Tech – I Semester Year: 2018-19

Branch : CSE Credits: 3

==========

1. Brief History and Scope of the Subject

C is a general-purpose, imperative computer programming language, supporting structured programming, variable scope and recursion, while a static type system prevents many unintended operations. By design, C provides constructs that map efficiently to typical machine instructions, and therefore it has found lasting use in applications that had formerly been coded in assembly language, including operating systems, as well as various application software for computers ranging from supercomputers to embedded systems.

C was originally developed by Dennis Ritchie in 1972 at Bell Laboratory, and used to re-implement the Unix operating system. It has since become one of the most widely used programming languages of all time, with C compilers from various vendors available for the majority of existing computer architectures and operating systems. C has been standardized by the American National Standards Institute(ANSI) since 1989 (see ANSI C) and subsequently by the International Organization for Standardization (ISO).

2. Pre-Requisites

- Introduction to computers

3. Course Objectives:

- To emphasize the use of flowcharts and pseudo code in problem solving.
- To gain knowledge in C language
- To apply C language in problem solving.

4. Course Outcomes:

Students will be able to

- design flowcharts and pseudo code for solving problems.
- understand C tokens and control statements.
- gain knowledge on arrays, strings, pointers, functions, structures and files.
- use C language for solving problems.
- self-learn advanced features of C.

5. Program Outcomes:

Graduates of the Computer Science and Engineering Program will have ability to Engineering Graduates will be able to:

1. **Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis:** Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

8. **Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **Life-long learning:** Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

6. Mapping of Course Outcomes with Program Outcomes:

	a	b	c	d	e	f	g	h	i	j	k	l
CO1	3	3	1	2								2
CO2	3	3	1	2								2
CO3	3	3	1	3								3
CO4	3	3	1	3				2				3
CO5	3	3	1	3							3	3

7. Prescribed Text Books

- Programming in C, Second Edition Pradip Dey and Manas Ghosh, OXFORD Higher Education.
- C Programming, E Balaguruswamy, 3rd edition, TMH.

8. Reference Text Books:

- Programming in C, Reema Thareja, OXFORD.
- C Programming, A Problem Solving Approach, Forouzan, Gilberg, Prasad, CENGAGE.
- R G Dromey, How to Solve it by Computer, Prentice-Hall of India, 1999.

9. URLs and Other E-Learning Resources

http://nptel.ac.in/courses/106104128/
http://nptel.ac.in/courses/106105085/
https://onlinecourses.nptel.ac.in/iitk_cs_101/

10. Digital Learning Materials:

http://www.learn-c.org/
http://www.tutorialspoint.com/cprogramming/

11. Lecture Schedule / Lesson Plan

Topic	No. of Periods	
	Theory	Tutorial
UNIT –1:		
PROBLEM SOLVING STEPS: Understanding problem, formulating a mathematical model	2	-
Solving the mathematical model	1	
Algorithm	1	2
Pseudo code and Flowchart, Coding, Testing and Debugging.	2	
General form of a C program, Identifiers, basic data types and sizes	2	2
Constants, variables, data modifiers, Arithmetic, relational and logical operators	2	
Variable declaration Statement and console I/O statements	2	2
Order of evaluation	1	
Simple problems such as evaluating formulae	1	
	14	**6**
UNIT – 2: Control Statements		
Selection statements: if, if-else, nested if, else-if, switch, nested switch and ? Operator	4	2
Iterative statements: Loops - while, do-while and for statements, Jump statements: break, return, go to, continue, exit()	3	
Problem solving**:** Factorial computation, generation of Fibonacci sequence, reversing digits of an integer, generating prime numbers.	3	2
	10	**4**
UNIT – 3: Arrays and Strings		
Declaring, initializing, accessing and display of one dimensional and two dimensional arrays.	2	2
Strings	2	
Problem solving: Computing mean and variance of a set of numbers , reverse the elements in an array, addition of two matrices.	4	2

	8	**4**
UNIT – 4:		
Pointers – Variables, Operators, Expressions and Multiple indirection	2	-
Functions – General form of functions	1	
Passing parameters by value and Passing parameters by address	2	2
Dynamic memory allocation functions, Pointers and arrays, Pointers and functions,	2	
Recursive functions and String handling functions	2	2
Problem solving : Print the sum of all elements of the array using pointers, swapping of two numbers, calculate the GCD of two non-negative integers using recursion.	2	
	11	**4**
UNIT – 5:		
Structures -Definition, declaration, initialization of structures	2	2
Accessing structure members, nested structures	1	
Arrays of structures, array within structures, structures and functions.	2	
Unions	1	2
Problem solving- Implement a structure to read and display the Name, Date of Birth and Salary of Employees, Functions to perform read, add and write two complex numbers using Structures.	1	
	7	**4**
UNIT – 6:		
File Handling- Text and binary files	1	2
file handling functions	1	
File Processing Operations – inserting, deleting, Searching and updating a record and displaying file contents.	1	2
Random access to files	1	
Problem solving – Copy the contents of one file to another, count the number of characters, words and lines in a file.	2	**2**
	6	**6**
Total No. of Periods:	**56**	**28**

UNIT – I

Objective:

- To introduce the steps of problem solving.

Syllabus:

Problem Solving Steps – Understanding problem, developing algorithm, flowchart, coding, debugging and testing.

General form of a C program, C Tokens, basic data types, type conversion, variable declarations, console I/O statements, order of evaluation.

Sample problems such as evaluating formulae.

Learning Outcomes:

At the end of the unit student will be able to

- understand the problem solving steps by using computer
- develop algorithms and logical flow charts for solving problems
- describe C tokens, basic data types, data modifiers, Type Coversion and console i/o statements.
- solve simple problems on computer using C

Learning Material

Problem solving steps: Problem solving is the process of solving a problem in a computer system by following a sequence of steps which include:

- Requirements Analysis
- Solving mathematical model(Design)
- Coding
- Debugging
- Testing

- **Requirements Analysis:**
  - **Understanding problem:** What is the objective of the problem

    Example: To calculate area of circle
  - **Analysis:**
    - **Input:** Radius value
    - **Output:** Area of circle
    - **Formulating mathematical model:** The formula **Area=3.14*r*r** can be used to generate the desired output.
    - **Data Elements:** r is used to store the input radius value and area is used to store the resultant value (area of circle).
- **Solving mathematical model(Design):**
  - **Developing an algorithm:** An algorithm is a sequence of steps written in the form of English phrases that specify the tasks that are performed while solving a problem.

    Algorithm for Area of circle:

    Step 1:Start

    Step 2: Read r

    Step 3: Compute Area=3.14*r*r;

    Step 4: Display Area

    Step 5: Stop

Characteristics of Algorithms:

- The instructions must be in an ordered form.
- The instructions must be simple and concise. They must not be ambiguous.
- There must be an instruction (condition) for program termination.
- The repetitive programming constructs must possess an exit condition otherwise, the program might run infinitely

- The algorithm must completely and definitely solve the given problem objective.

Advantages of algorithms:

- It provides the core solution to a given problem. This solution can be implemented on a computer system using any programming language of user's choice.
- It ensures easy comprehension of a problem solution as compared to an equivalent computer program.
- It eases identification and removal of logical errors in a program
- It facilitates algorithm analysis to find out the most efficient solution to a given problem.

Disadvantages of algorithms:

- In large algorithms, the flow of control becomes difficult to track.
- Algorithms lack visual representation of programming constructs like flowcharts; thus understanding the logic becomes relatively difficult.

Example 1: An Algorithm to add two integers and display the result:

Step 1: Start

Step 2: Read the first integer as input from the user.

Step 3: Read the second integer as input from the user

Step 4: Calculate the sum of the two integers

Sum = a+b

Step 5: Display sum as the result

Step 6: Stop

Example 2: An Algorithm to find out whether a given numbers is even or odd

Step 1: Start

Step 2: Accept a number from the user (num)

Step 3: if remainder of num divided by 2 (num/2) is zero then goto step 4

else go to step 5

Step 4: Display “number is an even number” and go to Step 6

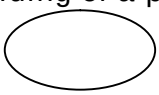
Step 5: Display “number is an odd number”

Step 6: Stop

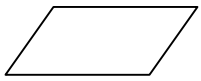
- **Flow Chart: Representing algorithm as logical flowchart:**
  - A flowchart is the graphical representation of the flow of control and logic in the solution of a problem. The flowchart is a pictorial representation of an algorithm
  - Flowcharts use different symbols for different activities which are performed at different stages of a process.

The various symbols used in a flowchart are:

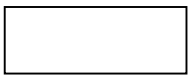
- **Start and end:** It is represented by an oval. It represents the starting and the ending of a process.



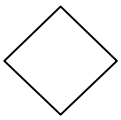
- **Input or output:** It is represented by a parallelogram. It represents the inputs given by the user to the process and the outputs given by the process to the user.



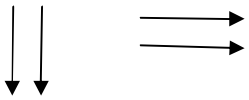
- **Action or process:** It is represented by a rectangle. It represents the actions, logics and calculations taking place in a process



- **Decision or condition:** It is represented by rhombus or a diamond shape. It represents the condition or the decision making step in a flowchart. The result of the decision is a Boolean value which is either true or false.

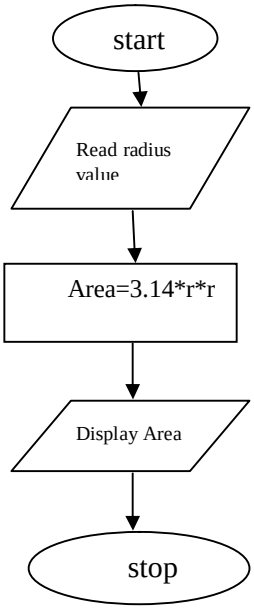


- **Arrow:** It is represented by a directed line. It represents the flow of process and the sequence of steps in a flowchart



- **Connectors:** It is represented by a circle in a flowchart. It represents the continuation of the flow of steps when a flowchart continues to the next page. A character such as an alphabet (a to z) or a symbol (α, β or □) etc can be placed in the circle.

Example: Flow chart for Area of Circle



Flowchart Design Rules:

- It must begin with a “start” and end with a “stop” symbol.
- The standard process flow should be either from top to bottom or from left to right.

- The arrows must be aligned properly so as to clearly depict the flow of program control.
- The use of connectors should be generally avoided as they make the program look more complex
- An action flowchart symbol must have only one input arrow and one output arrow

Advantages of flowcharts:

- It helps to understand the flow of program control in a easy way
- Developing program code by referring its flowchart is easier in comparison to developing the program from algorithm
- Its helps in avoiding semantic errors
- Any concept is better understood with the help of visual representation. It is easier to understand the pictorial representation of a programming logic.
- A flowchart acts as documentation for the process or program flow
- The use of flowcharts works well for small program design

Disadvantages of flowcharts:

- For a large program, the flowchart might become very complex and confusing
- Modification of a flowchart is difficult and requires almost an entire rework
- Since flowcharts require pictorial representation of programming elements, it becomes little tedious and time consuming to create a flowchart
- Excessive use of connectors in a flowchart may confuse the programmers

- **Representing algorithm as pseudo code:**
  - Pseudo code is much similar to algorithms.
  - It uses generic syntax for describing the steps that are to be performed for solving a problem along with the statement phrases for describing an action.

Example 1*:*

Pseudo code for area of circle

DEFINE: Real radius, Area

READ: Real radius

SET: Area=3.14*radius*radius

DISPLAY: Area

Example 2*:*

Pseudo code for addition of two numbers and display the result

DEFINE: Integer num1, num2, result

READ: Integer num1

READ: Integer num2

Set: result=num1+num2

DISPLAY: result

- **Coding:**
  - Developing a program refers to the process of writing source code for the required application following the syntax and the semantics of chosen programming language.
  - Syntax and semantics are the set of rules that a programmer needs to adhere while developing a program.
  - A programming language is typically bundled together with an IDE (Integrated Development Environment) containing the necessary tools for developing, editing, running and debugging a computer program.
  - Turbo C is provided with a strong and powerful IDE to develop, compile, debug and execute the programs.
  - C was evolved from ALGOL, BCPL and B by **Dennis Ritchie at the Bell Laboratories in 1972.**

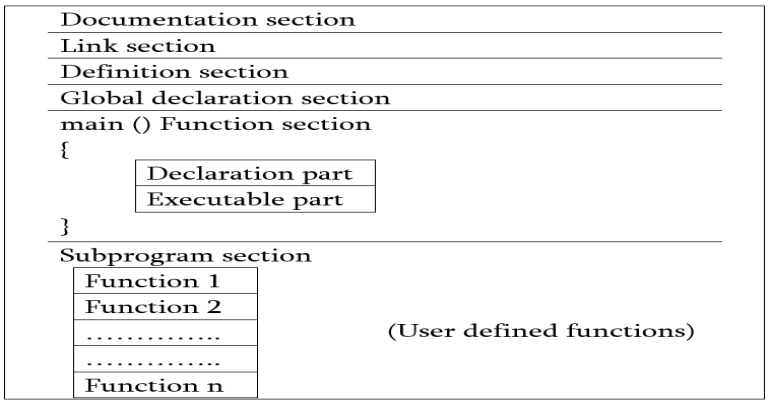
Importance of C:

- C is a robust language whose rich set of built-in functions and operators can be used to write any complex program.
- Programs written in C are efficient and fast. This is due to its variety of data types and powerful operators.

- There are only 32 keywords in ANSI C (American National Standards Institute) and its strength lies in its built-in functions.
- C is highly portable. This means that C programs written for one computer can be run on another with little or no modification.
- C language is well suited for structures programming, thus requiring the user to think of a problem in terms of function modules or blocks.

Basic structure of C program (or) General form of a C program:

- A C program can be viewed as a group of building blocks called functions.
- A function is a subroutine that may include one or more statements designed to perform a specific task.
- To write a C program, we first create functions and then put them together.
- A C program may contain one or more sections



- **Documentation section:** The documentation section consists of a set of comment lines giving the name of the program, the author and other details, which the programmer would like to use later.
- **Link section:** The link section provides instructions to the compiler to link functions from the system library.
- **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
  - **Declaration part:** The declaration part declares all the variables used in the executable part.
  - **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.
    - ❖ All statements in the declaration and executable part end with a semicolon.

- **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.

Note: All sections, except the main ( ) function section may be absent when they are not required.

➢ Debugging

- Debugging is used to detect the errors and bugs present in the programs.

- The debugger is a software that locates the position of the errors in the program code with the help of **Instruction set simulator (ISS)** technique.
- **ISS** is capable of stopping the execution of a program at the point where an erroneous statement is encountered.

A debugger perform the following tasks:

- Step-by-step execution of a program
- Stopping the execution of the program until the errors are corrected.

➢ **Testing:** The program must be tested with multiple input values to ensure that there are no logical errors present in the code.

Program (code):

```
#include<math.h>
#include<stdio.h>
#define pi 3.14
void main()
{
   float r,A;
   printf (“enter radius value”);
   scanf(“%2f”,&r);
   A=pi*pow(r,2);
   printf(“Area is %.2f”,A);
 }
```

Testing:

Enter radius value 4

Area is 50.24

Enter radius value 8

Area is 200.96

C Tokens:

The smallest individual unit in a C program is known as a token or a lexical unit.

C tokens can be classified as follows:

- Keywords
- Identifiers
- Constants
- Operators

C Keywords

- **C keywords** are the words that convey a special meaning to the c compiler.
- The keywords cannot be used as variable names because by doing so, we are trying to assign a new meaning to the keyword which is not allowed.

The list of C keywords is given below:

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			

C Identifiers

- Identifiers are used as the general terminology for the names of variables, functions and arrays.
- These are user defined names consisting of arbitrarily long sequence of letters and digits with either a letter or the underscore (_) as a first character.

There are certain rules that should be followed while naming c identifiers:

- They must begin with a letter or underscore (_).
- They must consist of only letters, digits, or underscore. No other special character is allowed.
- It should not be a keyword.
- It must not contain white space.
- Uppercase and lowercase letters are significant. That is the variable Total is not the same as total(or) TOTAL.
- It should be up to 31 characters long as only first 31 characters are significant.

Valid identifiers:

- Some ANSI C compilers do not consider two names to be different unless there is a variation within the first 31 characters.

Examples of valid identifiers:

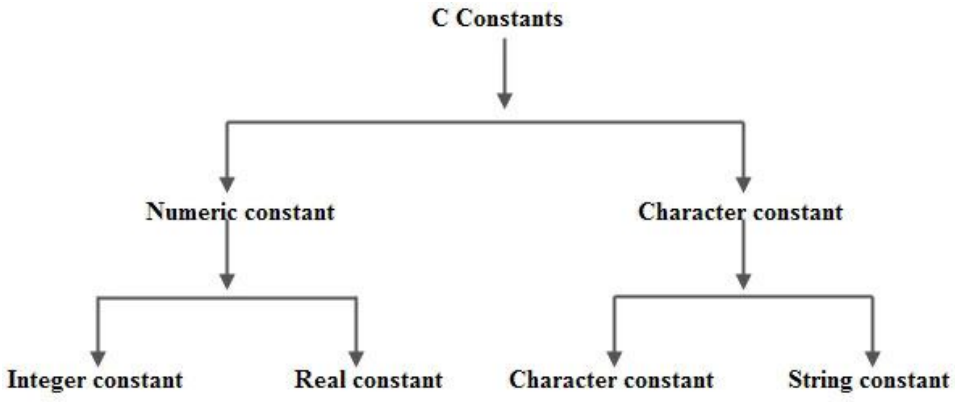
roll_number, marks, DeptCode

Examples of invalid identifiers:

23_student, %marks, @name, #emp_number, basic.pay, -HRA, (DA)

C Constants

- C constants refer to the data items that do not change their value during the program execution.



- **Integer constants:** An integer constant refers to a sequence of digits.

  There are three types of integers namely decimal integers, octal integers and hexadecimal integer

  - **Decimal integers** consists of a set of digits 0 to 9 preceded by an optional – or + sign.

    Valid examples of decimal integer constants are:

    123 -321 0 654321

    Embedded spaces, commas and non-digit characters are not permitted between digits.

    Invalid Examples: 15 750 20,000 $1000

  - **Octal integers** consist of any combination of digits from set 0 to 7, with a leading 0.

    Examples: 037 0 0435 0551

  - **Hexadecimal integers** are a sequence of digits preceded by 0x or 0X. This may also include alphabets A to F(represents numbers from 10 to 15).

    Examples: 0x2ef, 0X, 0XABD

- **Real constants:** Integer numbers are inadequate to represent quantities that vary continuously, such as distances, heights, temperatures and so on.

  These quantities are represented by numbers containing fractional parts like 17.548. Such numbers are called as real constants.

  Examples : 0.0083 -0.75 435.36 +247.0

  - A real number can be expressed in exponential notation.

    Mantissa e exponent

  - The mantissa is either a real number expressed in decimal notation or an integer

- The exponent is integer number with optional plus or minus sign.
    - The letter e separating the mantissa and the exponent can be written in either lowercase or uppercase i.e., 0.65e4 12E-2

- **Character constants**: A character constant consists of a single character enclosed in single quotes.

  Example : 'a'

- **String constants:** A string constant is a sequence of characters enclosed in double quotes.
    - So "a" is not the same as 'a'. the characters comprising the string constant are stored in successive memory locations
    - When a string constant is encountered in a C program, the compiler records the address of the first character and appends a null character('\0') to the string to mark the end of the string.
    - Thus the length of the string in constant is equal to number of characters in the string plus 1

      Example : length of "hello" is 6

C Operators

- C language supports a rich set of built-in operators.
- An operator is a symbol that tells the compiler to perform certain mathematical or logical manipulations.
- Operators are used in program to manipulate data and variables.

C operators can be classified into following types,

- Arithmetic operators
- Increment/Decrement operators
- Relation operators
- Logical operators
- Bitwise operators

- Assignment operators
- Conditional operator
- Special operators

Arithmetic operators:

C supports all the basic arithmetic operators. The following table shows all the basic arithmetic operators.

Operator	Description
+	adds two operands
-	subtract second operands from first
*	multiply two operand
/	divide numerator by denominator
%	remainder of division

Increment/Decrement operators:

The following table shows increment/decrement operators.

++	increment operator increases integer value by one
- -	Decrement operator decreases integer value by one

***Relational operators*:**

The following table shows all relational operators supported by C.

Operator	Description
= =	Check if two operands are equal
!=	Check if two operands are not equal.
>	Check if operand on the left is greater than operand on the right
<	Check operand on the left is smaller than right operand
>=	check left operand is greater than or equal to right operand
<=	Check if operand on left is smaller than or equal to right operand

***Logical operators*:**

C language supports following 3 logical operators. Suppose a=1 and b=0,

Operator	Description	Example
&&	Logical AND	(a && b) is false
| |	Logical OR	(a | | b) is true
!	Logical NOT	(! a) is false

***Bitwise operators*:**

Bitwise operators perform manipulations of data at **bit level**. These operators also perform **shifting of bits** from right to left. Bitwise operators are not applied to **float** or **double**.

Operator	Description
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
<<	left shift
>>	right shift

Truth table for bitwise **&, |** and **^**

a	b	a & b	a | b	a ^ b
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

- The bitwise shift operators shift the bit value.
- The left operand specifies the value to be shifted and the right operand specifies the number of positions that the bits in the value are to be shifted.
- Both operands have the same precedence.

Examples : a = 0001000 b= 2

a << b=0100000

a >> b=0000010

***Assignment operators*:**

Assignment operators supported by C language are as follows.

Operator	Description	Example
=	assigns values from right side operands to left side operand	a=b
+=	adds right operand to the left operand and assign the result to left	a+=b is same as a=a+b
-=	subtracts right operand from the left operand and assign the result to left operand	a-=b is same as a=a-b
*=	multiply left operand with the right operand and assign the result to left operand	a*=b is same as a=a*b
/=	divides left operand with the right operand and assign the result to left operand	a/=b is same as a=a/b
%=	calculate modulus using two operands and assign the result to left operand	a%=b is same as a=a%b

***Conditional operator*:**

It is also known as ternary operator and used to evaluate conditional expression.

expr1 ? expr2 : expr3

If **expr1** condition is true? Then value **expr2**: Otherwise value **expr3**

***Special operators*:**

Operator	Description	Example
sizeof	Returns the size of an variable	**sizeof(x)** return size of the variable **x**
&	Returns the address of the variable	**&x ;** returns address of the variable **x**
*	Pointer to a variable	***x ;** will be pointer to a variable **x**
,	Returns the value of the rightmost operand when multiple comma operators are used inside an expression.	**res=(num1,num2);** value of num2 will be assigned to variable res.

Basic data types in C:

- Data types specify how we enter data into our programs and what type of data we enter.
- C language has some predefined set of data types to handle various kinds of data that we use in our program.
- These data types have different storage capacities.

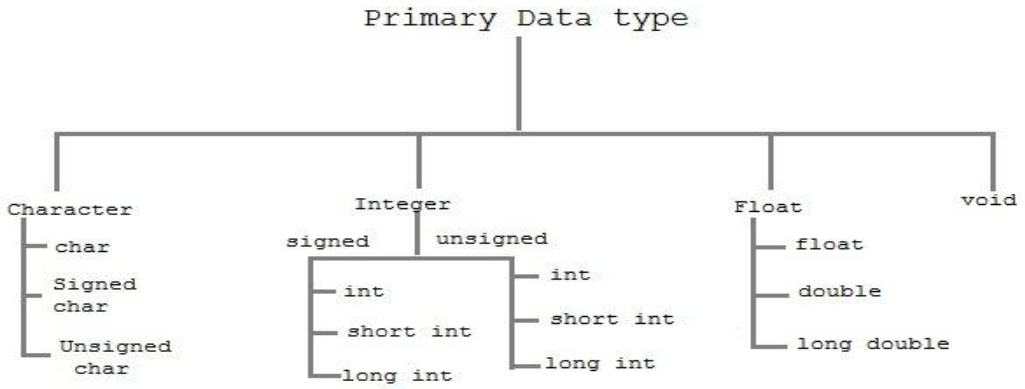
C language supports 2 different type of data types,

- **Primary data types**

  These are fundamental data types in C namely integer (**int**), floating(**float**), character (**char**) and **void**.

- **Derived data types**

  Derived data types are like arrays, functions, structures and pointers.



Integer type

Integers are used to store whole numbers.

Size and range of Integer type on 16-bit machine

Type	Size(bytes)	Range
int or signed int	2	-32,768 to 32767
unsigned int	2	0 to 65535
short int or signed short int	1	-128 to 127
long int or signed long int	4	-2,147,483,648 to 2,147,483,647
unsigned long int	4	0 to 4,294,967,295

Float type

Floating types are used to store real numbers.

Size and range of Integer type on 16-bit machine

Type	Size(bytes)	Range
Float	4	3.4E-38 to 3.4E+38
Double	8	1.7E-308 to 1.7E+308
long double	10	3.4E-4932 to 1.1E+4932

Character type

Character types are used to store characters value.

Size and range of Integer type on 16-bit machine

Type	Size(bytes)	Range
char or signed char	1	-128 to 127
unsigned char	1	0 to 255

void type

void type means no value. This is usually used to specify the type of functions.

Data modifiers in C

Modifiers are keywords in c which changes the meaning of basic data type in c. It specifies the amount of memory space to be allocated for a variable. Modifiers are prefixed with basic data types to modify the memory allocated for a variable.

Need of Data Modifiers:

- We use int to store the Salary of the employee as we are assuming that the salary will be in whole numbers.
- An integer data type takes 2 bytes of memory and we are aware that the Salary of any of the employee cannot be “Negative”.
- We are using “2 Bytes” to store the memory of an Employee and we can easily save 1 Byte over there by removing the “Signed Part” in the integer. This leads us to the user of Data Type Modifiers.

Types of Data Modifiers in C:

- Sign Modifier
- Size Modifier
- Const Modifier
- Volatile Modifier

1. Sign Modifier:

- All data types are “signed” by default. Signed modifier implies that the data type variable can store positive values as well as negative values.
- For example, if we need to declare a variable to store temperature, it can be negative as well as positive.

```
signed int temperature;      Or

int temperature;
```

- If we need to declare a variable to store the salary of an employee ,we will use “Unsigned” Data modifier.

```
unsigned int salary;
```

2. Size Modifier:

- Sometimes we need to increase the Storage Capacity of a variable so that it can store values higher than its maximum limit which is there as default.
- we need to make use of the “long” data type qualifier. “long” type modifier doubles the “length” of the data type when used along with it.
- For example, if we need to store the “annual turnover” of a company in a variable, we will make us of this type qualifier with 4 bytes in memory.

```
long int turnover;
```

- A “short” type modifier does just the opposite of “long” with 1 byte in memory. If one is not expecting to see high range values in a program and the values are both positive & negative.
- For example, if we need to store the “age” of a student in a variable, we will make use of this type qualifier as we are aware that this value is not going to be very high.

short int age;

Note: We can apply the above mentioned modifiers to character (char) base types.

* Variables of type char consume 1 byte in memory.

* Char can be signed or unsigned only.

* They have a range of -128 to 127 and 0 to 255 for signed & unsigned

respectively.

3. const modifier:

- In c all variables are by default not constant. Hence, you can modify the value of variable by program.
- You can convert any variable as a constant variable by using modifier const which is keyword of c language.

Properties of constant variable:

- You can assign the value to the constant variables only at the time of declaration.
  For example:
  ```
  const int i=10;
  float const f=0.0f;
  unsigned const long double ld=3.14L;
  ```

- Uninitialized constant variable is not cause of any compilation error. But you cannot assign any value after the declaration. For Example:
  ```
  const int i;
  ```
  If you have declared the uninitialized variable globally then default initial value will be zero in case of integral data type and null in case of non-integral data type. If you have declared the uninitialized const variable locally then default initial value will be garbage.

- Constant variables executes faster than non constant variables.

- You can modify constant variable with the help of pointers.

Example:

```
#include<stdio.h>
int main()
{
        const int i=10;
         int *ptr=&i;
        *ptr=(int *)20;
        printf("%d",i);
        return 0;
}
```

Output: 20

4. volatile modifier:

- we declare a variable as volatile every time the fresh value is updated.
- A volatile variable is declared with help of keyword volatile:

```
int volatile i;
```

- When any variable has qualified by volatile keyword in declaration statement then value of variable can be changed by any external device or hardware interrupt.
- If any variable has not qualified by volatile keyword in declaration statement, then compiler will take not volatile as default quantifier . What is meaning of the declaration:

```
const volatile int a=6;
```

Value of variable cannot be changed by program (due to const) but its value can be changed by external device or hardware interrupt (due to volatile).

Type conversion in C

A type conversion is a conversion from one type to another.
There are two types of type conversion.

- Implicit conversion
- Explicit conversion

Implicit type conversion

When an operator has operands of different types, they are converted to a common type according to semantic rules of a language.This is known as “automatic type conversion”, because it is done by the compiler on its own, without any external trigger from the user.

In general, the only automatic conversions are those that convert a “narrower” operand into a “wider” one without losing information.

Example: converting an integer to floating point value.

Expressions that might lose information, like assigning a floating-point type to an integer, may draw a warning, but they are not illegal.

Example :

```
#include<stdio.h>
main()
{
int a;
float b;
a=7.8;
b=150;
printf("a equals %d",a);
printf("\n b equals %f",b);
}
```

Output:

a equals 7
b equals 150.000000

Example 2

```
#include<stdio.h>
main()
{
int x=10;
char y='a';
float z;
x=x+y;          //y implicitly converted to int. ASCII value of 'a' is 97
z=x+1.0;
printf('x=%d,z=%f",x,z);
}
```

Output:

x=107, z=108.000000

Explicit type conversion:

This process is also called type casting and it is user defined.
Explicit type conversion can be forced ("coerced" in any expression, with a unary operator called a cast operator.
Syntax: (type-name) expression;

Example:

```
#include<stdio.h>
main()
{
float x=1.2;
int sum;
sum=(int)x+1;
printf("sum=%d",sum);
}
```

Output:
Sum=2

Variable declaration statement:

Each variable to be used in the program must be declared. To declare a variable, specify the data type of the variable followed by its name.

Declaration does two things:

- It tells the compiler what the variable name is.
- It specifies what type of data the variable will hold.

The declaration of variables must be done before they are used in the program

- Primary type declaration
- User-defines type declaration

A variable can be used to store a value of any data type. That is, the name has nothing to do with its type.

Syntax: datatype v1,v2,....vn

v1,v2,..vn are the names of variables. Variables are separated by commas. A declaration statement must end with a semi colon.

Example:

```
int count;

int number,total;

double ratio;
```

int and double are the keywords to represent integer type and real type data values respectively.

main() is the beginning of the program. The opening brace & signals the execution of the program.

User defined type declaration: C supports a feature known as " type definition" that allows users to define an identifier that would represent an existing data type.

Syntax:

```
typedef     type
```

Where type refers to an existing data type and "identifier" refers to the "new" name given to the datatype.

Example: typedef int units;

Console I/O statements:

- C language supports two formatting functions printf and scanf.

printf:

- The printf function (print formatting) is used to display information required by the user and also prints the values of the variables.
- It takes data values convert them to text stream using formatting functions specified in a control string and passes the resulting text stream to the standard output.

The syntax of printf function can be given as

```
int printf ("control string", variable list);
```

control string is the string that contains the text to be written to stdout.

The control string contains:

%c char single character.
%d (%i) int signed integer.
%e (%E) float or double exponential format.
%f float or double signed decimal.
%g (%G) float or double use %f or %e as required.

Return Value:

If successful, the total number of characters written is returned. On failure, a negative number is returned.

scanf:

- The function scanf() stands for scan formatting and is used to read formatted data from the keyword.
- The scanf function takes a text stream from the keyboard, extracts and formats data from the stream according to a format control string and then stores the data in a specified program variables.

The syntax of the scanf() can be given

```
int scanf("control string",arg1,arg2,arg3.....argn);
```

Example: int num;

scanf("%d",&num);

printf("%d",num);

Return Value:

If successful, the total number of characters written is returned, otherwise a negative number is returned. The return value of scanf function is the number of successful data inputs.

Reading a character:

- The simplest of all input/output operations is reading a character from the 'standard input' and writing it to the 'standard output' unit.
- Reading a single character is done by using **getchar().**

variable_name= getchar();

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

char name;

name=getchar();

This will assign the character 'H' to the variable name when we press the key H on the keyboard. Since it's a function, it its represented using parenthesis.

Writing a character:

Like getchar, there is an analogous function **putchar** for writing characters one at a time to the terminal.

putchar **(variable_name);**

where variable_name is a type char variable containing a character. This statement displays the character contained in the variable_name at the terminal.

answer='Y';

putchar(answer);

will display the character Y on the screen. The statement

putchar('\n');

would cause the cursor on the screen to move to the beginning of the next line.

Order of evaluation:

- C operators have two properties: priority and associativity.
- When an expression has more than one operator then based on priority, the operator which is having highest priority has to be

evaluated first. If two operators having same priority then consider the associativity of operators.

The below table explains about the list of operators that C language supports in the order of their precedence. The associativity indicates the order in which the operators of equal precedence in an expression are evaluated.

Symbol1	**Type of Operation**	**Associativity**
[ ] **( )** **.** **–>** postfix **++** and postfix **––**	Expression	Left to right
prefix **++** and prefix **– –** **sizeof** **&** * **+** **–** **~** **!**	Unary	Right to left
typecasts	Unary	Right to left
* **/** **%**	Multiplicative	Left to right

+ –	Additive	Left to right
<< >>	Bitwise shift	Left to right
< > <= >=	Relational	Left to right
= = !=	Equality	Left to right
&	Bitwise-AND	Left to right
^	Bitwise-exclusive-OR	Left to right
|	Bitwise-inclusive-OR	Left to right
&&	Logical-AND	Left to right
||	Logical-OR	Left to right
? :	Conditional-expression	Right to left
= *= /= %= += <<=	Simple and compound assignment2	Right to left

>>= &= ^= |=		
,	Sequential evaluation	Left to right

UNIT-I

Assignment-Cum-Tutorial Questions

Section-A

Objective Questions

1) An algorithm is an effective procedure for solving a problem in a finite number of steps. [True/False]

2) Which one is a valid identifier? [ ]

a) my_num b) 1my_num

c) my num d) $my_num

3) The size of double in bytes---------- [ ]

a) 2 b) 4 c) 10 d) 8

4) Which of the following is a string constant? [ ]

a) 'A' b) "A" c) ' ' d) '*'

5) Which of the following is not a floating point constant [ ]

a) 20 b) - 4.5 c) 'a' d) pi

6) Expression !0 < 2 is evaluated as [ ]

a) 0 b) 1 c) true d) false

9) Expression i = (2+3) *10 evaluation depends on [ ]

a) Associativity of ( ) operator b) Precedence of ( ) and * operator

c) both a and b d)None

10. What will be output of the following c program? [ ]

```
#include<stdio.h>
int main( )
{
        int _=5;
        int __=10;
            int ___;
         ___=_+__;
       printf("%i",___);
        return 0;
}
```

(A) 5 (B) 10 (C) 15 (D) Compilation error

11. The number of tokens in the following C statement. [ ]

printf(" The value of i = %d", i); is

a) 3 b) 7 c) 21 d)none

12. Value of x is [ ]

```
int  x=18 / 9 / 3 * 2 * 3 * 5 % 10;
```

a) 0 b) 1 c) 2 d) Compile time error

13. What is the output of the following program? [ ]

```
int main()
{
    int a = 1;
    int b = 1;
    int c = a || --b;
    int d = a-- && --b;
    printf("a = %d, b = %d, c = %d, d = %d", a, b, c, d);
    return 0;
}
```

a) a = 0, b = 1, c = 1, d = 0 c) a = 0, b = 0, c = 1, d = 0

b) c) a = 1, b = 1, c = 1, d = 1 d) a = 0, b = 0, c = 0, d = 0

14. What is the output of the program? [ ]

```
main()
{
    int x=7;
    x+=2;
    x=+2;
    printf("%d",x);
}
```

a) 2 b) 5 c) 7 d) compile error

15. What is the output of this C code? [ ]

```
#include<stdio.h>
int main()
{
int a=2,b=5;
a=a^b;
b=b^a;
printf("%d , %d",a,b);
return 0;
```

```
}
```

a) 7, 2 b) 2, 7 c) 7,7 d) 2,2

16.

```
#include <stdio.h>
int main()
{
    int i = 5, j = 10, k = 15;
    printf("%d ", sizeof(k /= i + j));
    printf("%d", k);
    return 0;
}
```

[ ]

Assume size of an integer as 4 bytes. What is the output of above program?

a)4 1 b)4 15 c)2 1 d)none

17. What is the output of this C code? [ ]

```
#include <stdio.h>
int main()
{
    int  i = (1, 2, 3);
    printf("%d", i);
    return 0;
}
```

a)1 b)3 c)Garbage value d)Compile time error

18. What is the output of this C code? [ ]

```
#include<stdio.h>
int main()
{
int x=10;
int y=20;
x+=y+=10;
printf("%d %d",x,y):
return 0;
}
```

Section-B

Descriptive Questions

1. Define algorithm. Draw flowchart to find the area of rectangle.

2. Give the structure of C program.

3. List out C tokens. Give examples for each.

4. Write short notes on data modifiers.

5. Explain about increment and decrement operators.

6. Give rules for order of evaluation.

7. Write an algorithm for finding roots of quadratic equation.

8. Draw a flowchart to calculate first year first semester percentage.

9. Write a program to swap two numbers without using third variable.

10. Evaluate the following expressions:

a) 8<<2 b) 1>3 && 4<5 c) 15>16 || 27>16 d) !10

SECTION-C

Gate Questions

1. Consider the following C program:

```
#include <stdio.h>
int main()
{
   int m = 10;
   int n, n1;
   n = ++m;
   n1 = m++;
   n --;
   --n1;
   n -= n1;
   printf("%d",n);
   return 0;
}
```

The output of the program is _____. **GATE CS 2017 (SET 2)**

2. The attributes of three arithmetic operators in some programming language are given below.

Operator	Precedence	Associativity	Arity
+	High	Left	Binary
−	Medium	Right	Binary

*	Low	Left	Binary

The value of the expression 2 − 5 + 1 − 7 ∗ 3 in this language is----------. **GATE CS 2016**

3. The number of tokens in the following C statement is _____.

[ ]

```
printf("i = %d, &i = %x", i, &i);
```

(a) 3 (b) 26 (c) 10 (d) 21 **GATE CS 2000**

UNIT-2

Objective:

- Familiarize about control statements in C language.

Syllabus:

Control Statements:

Selection Statements – if, if-else, nested if, else-if, switch and conditional operator.

Iteration Statements – for, while and do-while.

Jump Statements – return, goto, break, exit and continue.

Problem Solving - Factorial computation, generation of Fibonacci sequence, reversing digits of an integer, generating prime numbers.

Learning Outcomes:

At the end of the unit student will be able to:

- understand the control statements in C.
- solve moderate problems on computer using control statements in C.

CONTROL STATEMENTS:

Controlling the flow of program is very important aspect of programming. Control flow relates to the order in which the operations of a program executed.

Two types of control structures:

- Selection / Branching Statements
- Iteration / loop Statements

SELECTION STATEMENTS:

A selection statement selects among a set of statements depending on the value of a controlling expression.

The selection statements are:

- if statement
- switch statement

C has four kinds of if statements that permit the execution of a single statement or a block of statements based on the evaluation of a test expression. The statements are

1. if
2. if-else
3. if-else-if
4. nested if

1. if Statement:

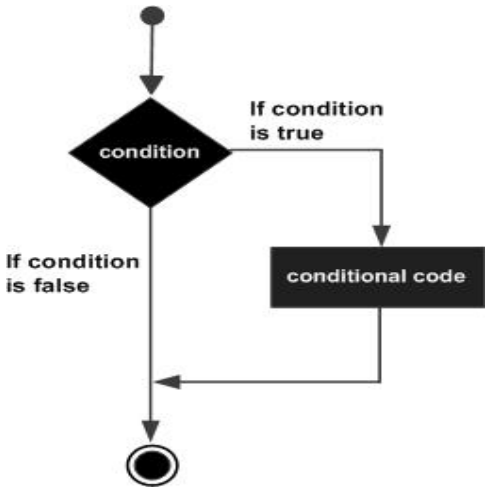
An **if** statement consists of a Test expression followed by one or more statements. It is a one-way decision statement.

Syntax:

```
if(Text Expr)
{
        /*statement(s) will execute if the Test expression is true*/
}
```

- If the Test expression evaluates to **true**, then the block of code inside the 'if' statement will be executed.
- If the Test expression evaluates to **false**, then the first set of code after the end of the 'if' statement (after the closing curly brace) will be executed.

Flow Diagram



Example

```
#include<stdio.h>
void  main()
{
        int a=10;       /*local variable declaration*/
        if(a<20)
```

```
    {
        printf("a is less than 20\n");
    }
    printf("The value of a :%d",a);
}
```

When the above code is compiled and executed, it produces the following result −

a is less than 20

The value of a: 10

C programming language assumes any non-zero and non-null values as true, and if it is either zero or null, then it is assumed as false value.

2. if-else statement:

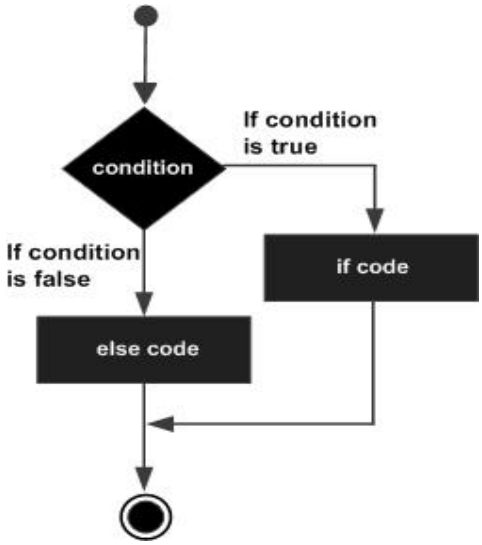
It is a two-way decision statement. Similar to one-way decision, the decision here is based on the test expression.

Syntax

```
if(Text Expr)
{
    /*statement(s) will execute if the Test expression is true*/
}
else
{
    /*statement(s) will execute if the Test expression is false*/
}
```

- If the Test expression evaluates to **true**, then **if block** will be executed.
- Otherwise, **else block** will be executed.

Flow Diagram



Example

```
#include<stdio.h>
void main()
{
        int a=10;       /*local variable declaration*/
        if(a<20)
        {
                printf("a is less than 20\n");
        }
        else
        {
        printf("a is not less than 20\n");
        }
}
```

When the above code is compiled and executed, it produces the following result

a is less than 20

3. if-else-if Statement:

It is a multi-way decision statement. These are used to test various conditions.

When using if-else-if statements, there are few points to keep in mind −

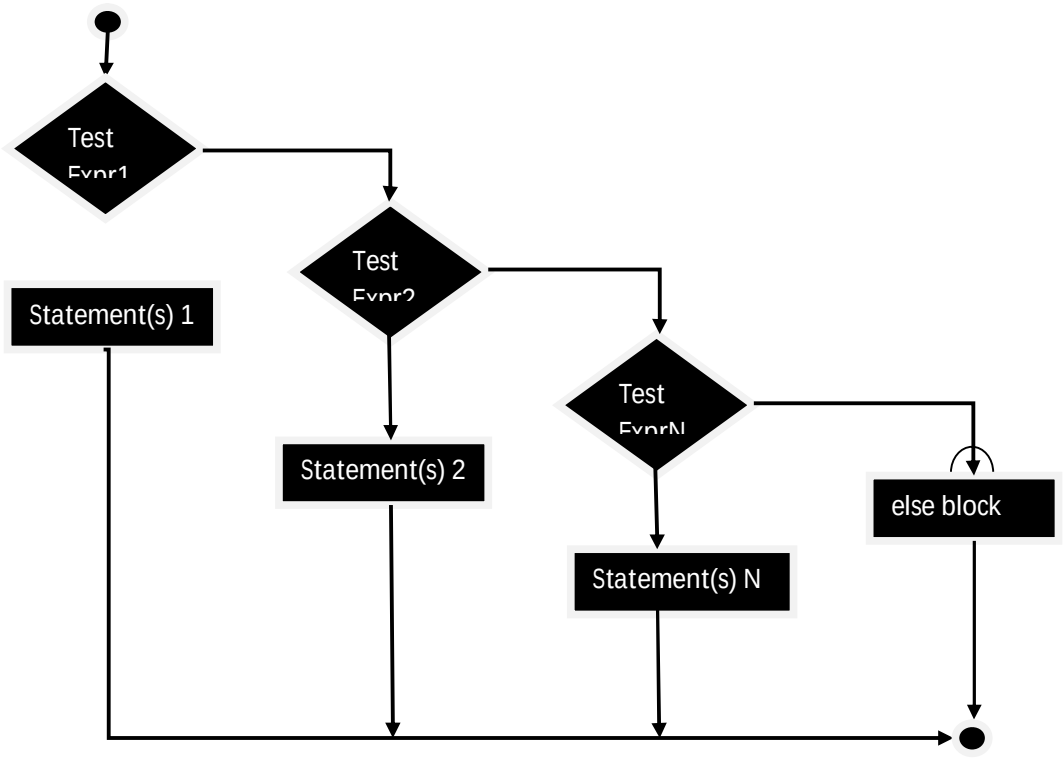
- An if can have zero or one else's and it must come after any else if's.

- An if can have zero to many else if's and they must come before the else.
- Once an else if succeeds, none of the remaining else if's or else's will be tested.

Syntax

```
if(Test Expr1)
{
        /*statement(s) will execute if the Test expression1 is true*/
}
else  if(Test Expr2)
{
        /*statement(s) will execute if the Test expression2 is true*/
}
                .
                .
else  if(Test ExprN)
{
        /*statement(s) will execute if the Test expressionN is true*/
}
else
{
        /*executes when the none of the above condition is true*/
}
```

Flow Diagram



Example

```
#include<stdio.h>
void  main()
{
        int a=100;      /*local variable declaration*/
        if(a= =20)
        {
                printf("Value of a is 20\n");
        }
        else  if(a==30)
        {
                printf("Value of a is 30\n");
        }
        else  if(a==40)
```

```
        {
                printf("Value of a is 40\n");
        }
        else
        {
                printf("None of the values is matching");
        }
}
```

When the above code is compiled and executed, it produces the following result

None of the values is matching

4. Nested if statement

- When any if statement is written under another if statement, this cluster is called nested if.
- It is always legal in C programming to nest if-else statements, which means you can use one if or else if statement inside another if or else if statement(s).

Syntax

```
if(Test Expr1)
{
        /*Statements will execute if the test expr 1 is true*/
        if(Test Expr2)
        {
                /*Statements will execute if both test expr1 and test expr2 is true*/
        }
}
```

Example

```
#include<stdio.h>
void  main ( )
{
        int   a,  b, c;
        printf ("\n enter  three numbers :");
```

```
scanf ("%d%d%d ", &a,&b,&c);
if(a > b)
{
        if(a>c)
        {
                printf("\n a is big");
        }
        else
        {
                printf("\n c is big");
        }
        }
        else
        {
                if (b > c)
                {
                        printf("\n b is big");
                }
                else
                {
                        printf("\n c is big");
                }
        }
```

When the above code is compiled and executed, it produces the following result :

Enter three numbers:

2 6 4

b is big

Dangling else problem:

In nested if statements, when a single "else clause" occurs, the situation happens to be dangling else. For example:

```
if(condition)
        if(condition)
                if(condition)
else
        printf("dangling else!");
```

1. In such situations, else clause belongs to the closest if statement which is incomplete that is the innermost if statement.
2. However, we can make else clause belong to desired if statement by enclosing all if statements in block outer to which if statement to associate the else clause.

For example:

```
if(condition)
{
    if(condition)
        if(condition)
}
else
printf("\n else associates with the outermost if statement");
```

switch statement:

A switch case statement is a multi-way decision statement that is simplified version of if-else block that evaluates only one variable.

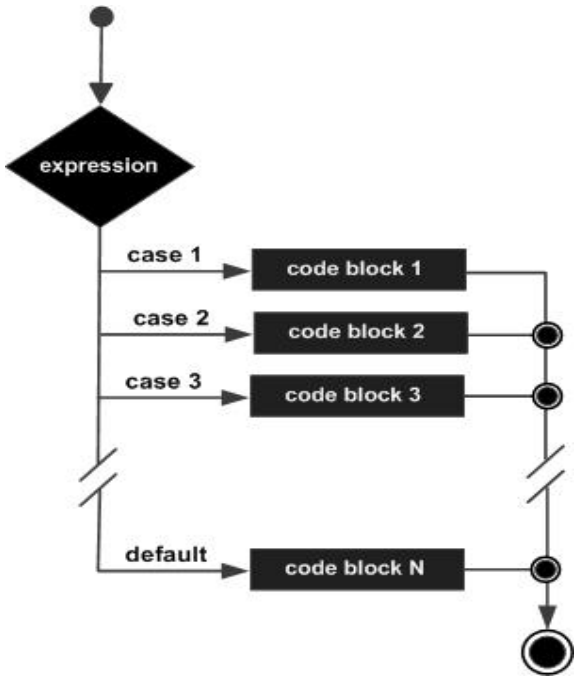
Syntax:

```
switch (expr)
{
    case constant1:   stmtList1;
                      break;
    case constant2:   stmtList2;
                      break;
    ----------
    ----------
    default:      stmtListN;
}
```

The following rules apply to a switch statement −

- The expression used in a switch statement must have an integral or enumerated type, or be of a class type in which the class has a single conversion function to an integral or enumerated type.
- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- The constant-expression for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.
- When the variable being switched on is equal to a case, the statements following that case will execute until a break statement is reached.
- When a break statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
- Not every case needs to contain a break. If no break appears, the flow of control will fall through to subsequent cases until a break is reached.
- A switch statement can have an optional default case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

Flow Diagram



Example

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int a,b,ch,c;
        clrscr();
        printf("\nenter a and b values");
        scanf("%d%d",&a,&b);
        printf("\n1.Addition\t2.Subtraction\t3.Multiplication\t4.Division")
;
        scanf("%d",&ch);
        switch(ch)
        {
          case 1:          c=a+b;
                   break;
         case 2: c=a-b;
                   break;
                    case 3: c=a*b;
                   break;
         case 4: c=a/b;
                   break;
         default:printf("\nenter valid choice");
        }
        printf("\nresult=%d",c);
          getch();
}
```

When the above code is compiled and executed, it produces the following result

```
enter a and b values
5 6
1. Addition     2 .Subtraction
3.Multiplication        4.Division  1
result=11
```

Nested Switch:

It is possible to have a switch as a part of the statement sequence of an outer switch. Even if the case constants of the inner and outer switch contain common values, no conflicts will arise.

Syntax:

```
switch(ch1)
{
        case  'A':  printf("This A is part of outer switch");
        switch(ch2)
        {
                case 'A':     printf("This A is part of inner switch");
                              break;
                case 'B':     /*case code*/
        }
                case 'B':   /*case code*/
}
```

Conditional Operator:

Conditional operator or the ternary(?:) is just like an if-else statement that can be within expressions. Such an operator is useful in situations in which there are two or more alternatives for an expression.

Syntax:

```
exp1? exp2 : exp3
```

exp1 is evaluated first. If it is true, then exp2 is evaluated and becomes the result of the expression. Otherwise exp3 is evaluated and becomes the result of the expression.

Example:

```
large=(a>b)?a:b;
```

ITERATION STATEMENTS:

Iterative statements are used to repeat the execution of a list of statements, depending on the value of an integer expression.

C language supports three types of iterative statements also known as looping statements. They are:

- while loop
- do-while loop
- for loop

1. while loop

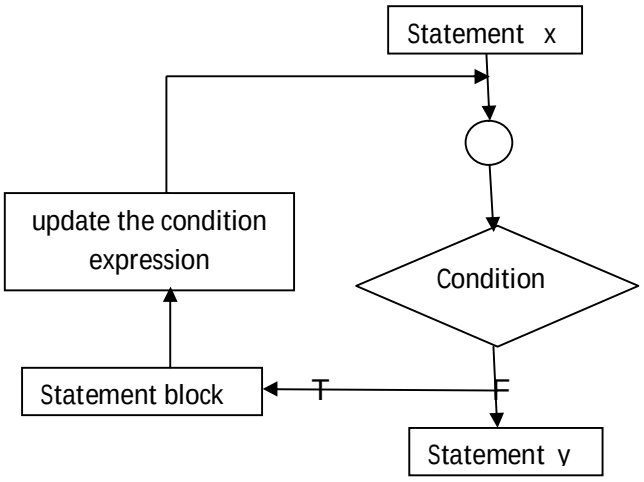
The while loop provides a mechanism to repeat one or more statements while a particular condition is true.

Syntax:

```
statement x;
while(condition)
{
        statement block;
}
statement y;
```

- In the while loop, the condition is tested before any of the statements in the statement block is executed.
- If the condition is true only the statement block will be executed otherwise (if the condition is false), the control will jump to statement y, which is the immediate statement outside the while loop block.

Flow Diagram:



Example

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int a=1;
        clrscr();
        while(a<=5)
        {
                printf("%d\t",a);
                a++;
        }
}
```

When the above code is compiled and executed, it produces the following result

1 2 3 4 5

2. do-while loop

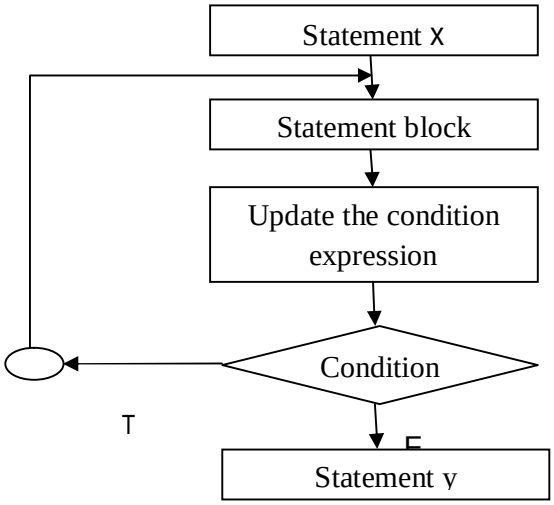
The do-while loop is similar to the while loop. The only difference is that in a do-while loop, the test condition is tested at the end of the loop. Now the test condition is tested at the end, this clearly means that the body of the loop gets executed at least one time even if the condition is false.

Syntax:

```
statement x;
do
{
        statement block;
}while(condition);
statement y;
```

Disadvantage: The major disadvantage of using a do-while loop is that it always executes at least once, even if the user enters some invalid data.

Flow Diagram



Example

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int i=1;
        clrscr( );
        do
        {
                printf("%d\t",i);
                i++;
        }while(i<=5);
}
```

When the above code is compiled and executed, it produces the following result

1 2 3 4 5

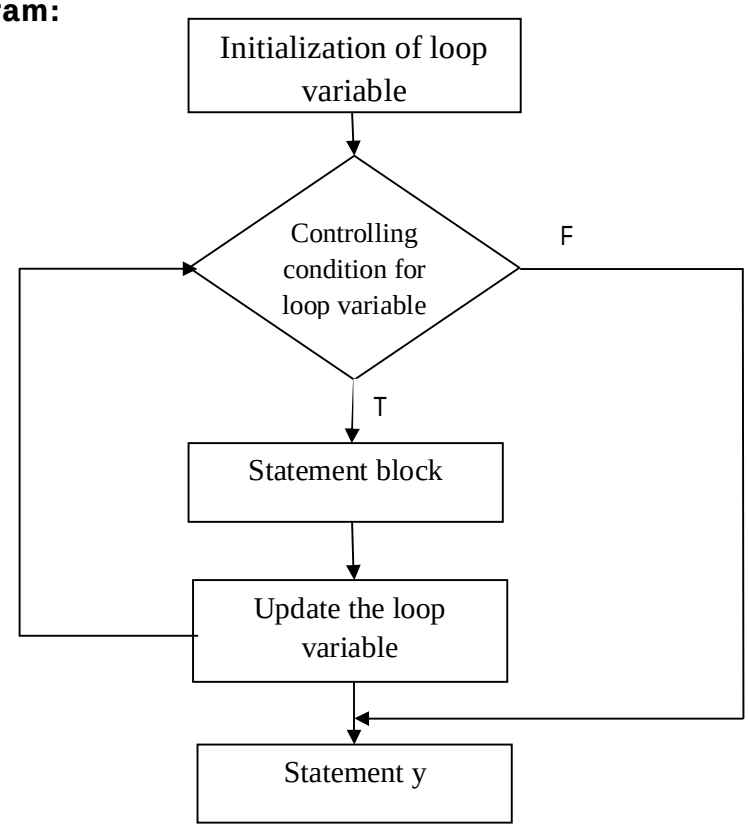
3. for loop

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

Syntax:

```
for( initialization; condition; increment/decrement/update)
{
        statement block;
}
statement y;
```

Flow Diagram:



Here is the flow of control in a 'for' loop −

- The **initialization** step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.
- Next, the **condition** is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and the flow of control jumps to the next statement just after the 'for' loop.
- After the body of the 'for' loop executes, the flow of control jumps back up to the **increment** statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the condition.
- The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes false, the 'for' loop terminates.

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int i;
        clrscr();
        for( i=1; i<=5; i++)
        {
                printf("%d\t",i);
         }
}
```

When the above code is compiled and executed, it produces the following result

```
1       2       3       4       5
```

Nested loops

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

Syntax for nested for loop

```
for ( initialization; condition; increment / decrement / update)
{
        for  ( initialization; condition; increment / decrement / update)
        {
                statement(s);
        }
         statement(s);
}
```

Syntax for nested while loop

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

```
}
```

Syntax for nested do-while loop

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

Note: you can put any type of loop inside any other type of loop. For example, a 'for' loop can be inside a 'while' loop or vice versa.

JUMP STATEMENTS:

Jump statements cause an unconditional jump to another statement elsewhere in the code. They are used primarily to interrupt switch statements and loops. The jump statements are:

1. goto statement
2. break statement
3. continue statement
4. return statement
5. exit statement

1. goto statement :

The goto statement is used to transfer control to a specified label. However, the label must reside in the same function and can appear only before one statement in the same function.

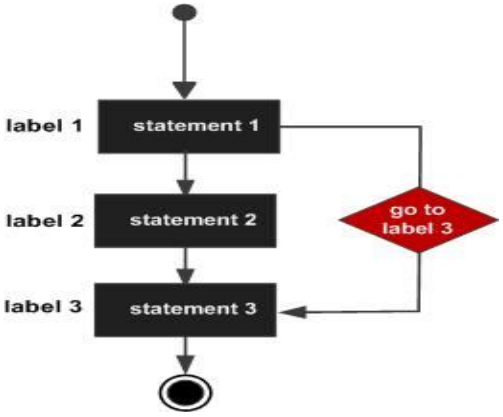
Syntax:

```
goto label;
...
...
label:
statement(s);
```

- The label can be placed anywhere in the program either before or after the goto statement.

- If the label is placed after the goto statement, then it is called a forward jump and in case it is located before the goto statement , it is said to be a backward jump.

Flow Diagram



2.break statement :

The **break** statement in C programming has the following two usages −

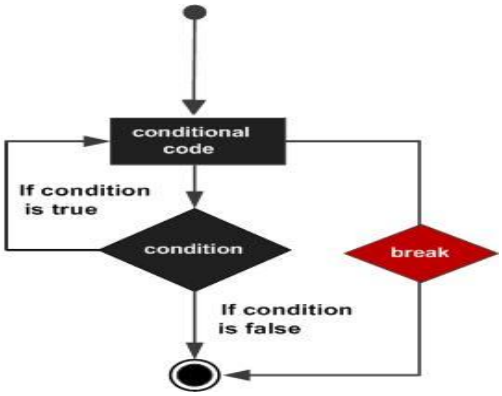
- When a **break** statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.
- It can be used to terminate a case in the **switch** statement.

If you are using nested loops, the break statement will stop the execution of the innermost loop and start executing the next line of code after the block.

Syntax

```
break;
```

Flow Diagram



Example

```
#include<stdio.h>
void main()
{
        int i=0;
        while(i<=10)
        {
                if(i= = 5)
                {
                        break;
                }
                printf("\t%d",i);
                i=i+1;
        }
}
```

When the above code is compiled and executed, it produces the following result

0 1 2 3 4

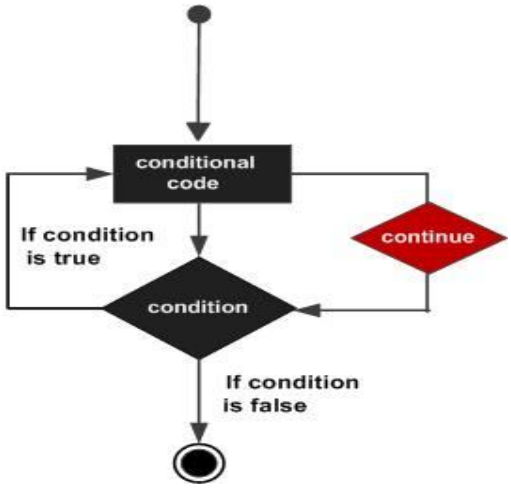
3. continue statement :

The **continue** statement in C programming works somewhat like the **break** statement. Instead of forcing termination, it forces the next iteration of the loop to take place, skipping any code in between.

Syntax :

```
continue;
```

Flow Diagram



Example

```
#include<stdio.h>
void main()
{
        int i=0;
        while(i<10)
        {
                i=i+1;
                if(i= = 5)
                continue;
                printf("\t%d",i);
        }
}
```

When the above code is compiled and executed, it produces the following result

1 2 3 4 6 7 8 9 10

4. return Statement :

The return statement terminates execution of a function and returns control to the calling function, with or without a return value. A function may contain any number of return statements.

Syntax:

```
return;
```

Reaching the closing brace that terminates a function is equivalent to executing a return statement without an expression.

5. exit statement :

The exit statement terminates the program. File buffers are flushed, streams are closed, and temporary files are deleted.

Syntax:

```
void exit(int status);
```

Status	Description
0	Successful termination
1	Unsuccessful termination

The required header for the exit function is:

```
#include <stdlib.h>
```

UNIT-II

Assignment-Cum-Tutorial Questions

Section-A

Objective Questions

1. No two case labels can have the same value. [True/False] [ ]
2. Based on the given statements select the most appropriate option [ ]

Statement I: do..while statement is an entry controlled loop.

Statement II: continue statement is used to go to the next iteration in a loop

a. I&II true b. only I is true c. Only II is true d. Both are false

3. In a for loop, if the condition is missing, then? [ ]

a. it is assumed to be present and taken to be false

b. it is assumed to be present and taken to be true

c. it result in the syntax error

d. execution will be terminated abruptly

4. Which of the following statement is used to take the control to the beginning of the loop [ ]

a. exit b. continue c. break d. None

5. The _____ statement is used to transfer control to a specified label.
6. A do-while loop is useful when we want the statement within the loop must be executed? [ ]

a. Only once b. At least once c. More than once d.None of the above

7. Based on the given statements, select the most appropriate option [ ]

Statement I : break statement is used to terminate from the program.

Statement II : for statement is an entry controlled loop.

a. I&II true b. only I is tru c. Only II is true d. Both are false

8. Which of the following cannot be checked in a switch-case statement? [ ]

a. char b. int c. float d. enum

9. The following program fragment [ ]

```
if(a=7)
```

```
        printf(“ a is seven”);
    else
        printf(“ a is not seven”);
```

results in the printing of

a. a is seven b. a is not seven c. nothing d. garbage

10. The output of the code below is [ ]

```
#include <stdio.h>
 void  main()
 {
    int x = 0;
    if (x == 0)
            printf("hi");
    else
        printf("how are u");
         printf("hello");
}
```

a. hi b. how are you c. hello d. hihello

11. The output of the code below is(when 1 is entered) [ ]

```
#include <stdio.h>
void  main()
{
        double ch;
        printf("enter a value btw 1 to 2:");
        scanf("%lf", &ch);
         switch (ch)
         {
            case 1: printf("1");
                    break;
            case 2: printf("2");
                     break;
        }
```

```
}
```

a. Compile time error b. 1 c. 2 d. None of the above

12. The following program fragment results in [ ]

```
int i=107,x=5;
printf((x<7)?"%d":"%c",i);
```

a. an execution error b. a syntax error

c. printing of k d. none of the above

13. The following statements will result in the printing of [ ]

```
for( i=3; i<15; i+=3 )
{
  printf("%d",i);
  ++i;
}
```

a. 3 6 9 12 b. 3 6 9 12 15 c. 3 7 11 d. 3 7 11 15

14. What is the output of this C code? [ ]

```
#include <stdio.h>
const int a = 1,  b = 2;
int main()
{
        int x = 1;
        switch (x)
       {
              case a: printf("yes ");
              case b: printf("no\n");
                      break;
        }
}
```

a) yes no b) yes c) no d) Compile time error

15. What is the output of this C code? [ ]

```
#include <stdio.h>
 int main()
{
        do
          printf("In while loop ");
        while (0);
printf("After  loop\n");
}
```

a) In while loop b) In while loopAfter loop c) After loop d) Infinite loop

16. How many times "India" is get printed? [ ]

```
#include<stdio.h>
int main()
{
   int x;
   for(x=-1; x<=10; x++)
   {
      if(x < 5)
         continue;
      else
         break;
      printf("India");
   }
   return 0;
}
```

A. Infinite times B. 11 times C. 0 times D.10 times

17. What is the output of the code given below? [ ]

```
int main()
 {
    printf("%d ", 1);
     goto l1;
     printf("%d ", 2);
     l1:goto l2;
     printf("%d ", 3);
     l2:printf("%d ", 4);
}
```

A. 1 4 B. Compilation error C. 1 2 4 D. 1 3 4

Section-B

Descriptive Questions

1. Define multi way selection? Explain switch statement with syntax.
2. When dangling else problem occurs? Explain.
3. Explain for loop structure with sample code.
4. Explain various Iterative statements in C language.
5. Differentiate break and continue statements
6. Write the difference between while and do while.
7. Illustrate various Conditional statemnts in C language.

Programs:

8. Write a c program to find the roots of quadratic equation.
9. Write a program to calculate electricity bill based on the assumed constraints.
10. Write a program to generate all the prime numbers between 1 and n.
11. Write a c-program to print Fibonacci series
12. Write a program to find the LCM and GCD of given two numbers.
13. Write a program to find the reverse of the given integer.

SECTION-C

Questions asked in Competative exams

1. Consider the following program

```
# include<stdio.h>
int main ( )
{
            int i, j, k = 0;
            j = 2 * 3 / 4 + 2.0 / 5 + 8 / 5;
            k – = – –j;
            for (i = 0; i < 5: i++)
           {
                 switch (i + k)
                {
               case 1:
               case 2:   printf (“\ n%d”, i+k);
               case 3:   printf (“\n%d”, i+k);
               default:  printf (“\n%d”, i+k);
               }
           }
             return 0;
  }
```

The number of times printf statement is executed is _____.

Gate CS 2015

2. What will be the output of the following C program segment? [ ]

```
char inchar = 'A';

switch (inchar)

{

    case 'A' :    printf ("choice A") ;

    case 'B' :    printf ("choice B ") ;

    case 'C' :

    case 'D' :

    case 'E' :

    default:   printf ("No Choice") ;

}
```

(a) No choice (b) Choice A

(c) Choice AChoice B No choice (d) Program gives no output as it is erroneous

GATE CS 2012

3. Consider line number 3 of the following C-program. [ ]

```
int main ( ) { /* Line 1 */
  int i, n; /* Line 2 */
  for (i =0, i<n, i++); /* Line 3 */
}
```

Identify the compiler's response about this line while creating the object-module:

(a) No compilation error (b) Only a lexical error

(c) Only syntactic errors (d) Both lexical and syntactic errors

GATE CS 2005

4. Consider the following C program [ ]

```
main()
{
  int x, y, m, n;
  scanf ("%d %d", &x, &y);
  /* x > 0 and y > 0 */
  m = x; n = y;
  while (m != n)
  {
    if(m>n)
      m = m - n;
    else
      n = n - m;
  }
  printf("%d", n);
}
```

The program computes

(a) x + y using repeated subtraction

(b) x mod y using repeated subtraction

(c) the greatest common divisor of x & y

(d) the least common multiple of x & y

GATE CS 2004

UNIT III

Objective:

- Impart skills in solving problems using arrays and strings.

Syllabus:

Arrays and Strings– Declaring, initializing, accessing and display of one dimensional and two dimensional arrays.

Problem Solving – Computing mean and variance of a set of numbers, reverse the elements in an array, addition of two matrices

Learning Outcomes:

Students will be able to

- understand the basic concept of arrays
- describe types of arrays
- solve moderate problems using arrays

Learning Material

Arrays

- An array is a collection of **elements of same data type**.
- An array element can be referenced by an **integer index** starting from 0 to the size of array minus 1.
- The array elements are stored in **continuous memory locations**, the lowest address corresponds to the first element and the highest address to the last element.

Types of Arrays

- One-dimensional arrays
- Multi-dimensional arrays

- C supports **one-dimensional** and **multi-dimensional** arrays.
- The simplest form of the multi-dimensional array is the **two-dimensional array**.
- One-dimensional array is like a **list** and two-dimensional array is like a **table**.

One-Dimensional Arrays

- A one-dimensional array is a list of items given by **one variable** name using only **one subscript**.

1. Declaring 1-D Arrays

- To declare an array in C, **name** of the array, **type** of elements (valid data type in C) and total **number** of elements (positive integer) must be specified as follows:

Syntax: Type arrayName [arraySize];
Eg: int number[10];

2. Initializing 1-D Arrays

- One-Dimensional arrays can be initialized during **compile-time** or **run-time**.

2.1 Compile-Time Initialization

- Whenever we declare an array, we initialize that array directly at compile time.
- Initializing an 1-D array is called as Compile-Time Initialization if and only if we assign certain set of values to array elements before executing the program.
- The values of the elements must be enclosed between floral brackets {}, if more than one element value is provided.
- There are two different ways of compile time initialization of an array.

2 Ways of Compile Time Initialization of 1-D Array

1. Size is Specified Directly
2. Size is Specified Indirectly

Method 1 : Array Size Specified Directly

- In this method , we try to specify the Array Size directly.

int num[5] = { 2 , 8 , 7 , 6 , 0 };

- In the above example we have specified the size of array as 5 directly in the initialization statement.
- Compiler will assign the set of values to particular element of the array.

num[0] = 2

num[1] = 8

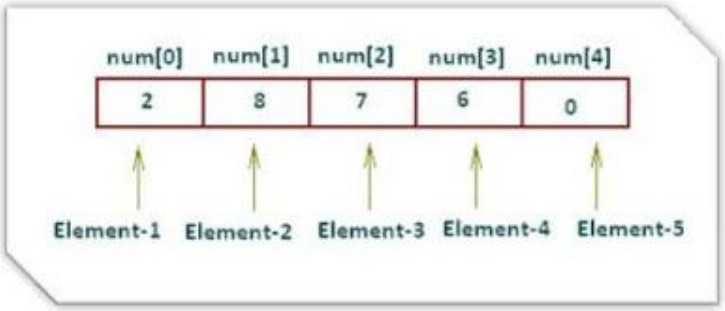
num[2] = 7

num[3] = 6

num[4] = 0

- As at the time of compilation all the elements are at Specified Position, so this Initialization Scheme is Called as "**Compile Time Initialization**".

Graphical Representation



Method 2 : Size Specified Indirectly

- In this scheme of compile time Initialization, we do not provide size to an array but instead we provide set of values to the array.

```
int     num[ ] = { 2 , 8 , 7 , 6 , 0 };
```

- Compiler counts the number of elements written inside pair of braces and determines the size of an Array.
- After counting the number of elements inside the braces, the size of array is considered as 5 during complete execution.
- This type of Initialization Scheme is also Called as "**Compile Time Initialization**"

2.2 Run-Time Initialization

- This is standard way of initialization.
- User will be able to set their value while execution using **scanf( )** function.
- In the run time initialization of the arrays, **looping statements** are almost compulsory.
- Looping statements are used to initialize the values of the arrays one by one by using assignment operator or through the keyboard by the user.

```
for(i = 0 ; i < 10 ; i++)
    {
        scanf(" %d ", &x[ i ] );
    }
```

- Above example will initialize array elements with the values entered through the keyboard.

Points to remember

- The no of elements cannot be greater than the size of array.
- If the no of values is less than size, remaining element values are initialized with default values of the data type.

- The array values can be given manually, dynamically (user input) or by arithmetic operations.

3. Accessing 1-D Array Elements

- An array element can be accessed using the array index.
- This is done by placing the index of the element within square brackets after the name of the array.

```
int number[ 10 ] = { 10 , 20 , 30 , 40 , 50 , 60 , 70 , 80 , 90 , 100 };
int n = number[ 7 ];
```

- The above statement will take the 7th element from the array and assign the value to "n" variable.

Example Program

```
#include <stdio.h>
int main()
{
int num[ ] = { 2 , 8 , 7 , 6 , 0 };
int i;
for(i = 0 ; i < 5 ; i++) {
   printf("\nArray Element num[ %d ] : %d", i+1 , num[ i ]);
}
return 0;
}
```

Output

```
Array Element num [ 1 ] = 2
Array Element num [ 2 ] = 8
Array Element num [ 3 ] = 7
Array Element num [ 4 ] = 6
Array Element num [ 5 ] = 0
```

Two-Dimensional Array

- The simplest form of the multi-dimensional array is the two-dimensional array.
- A two-dimensional array is a list of items given by **one variable** name using **two subscripts** (number of rows and number of columns).

1. Declaring 2-D Arrays

- To declare a two-dimensional integer array with X number of rows and Y number of columns:

Syntax: **type** **arrayName[X] [Y];**

- A two-dimensional array can be considered as a table which will have X number of rows and Y number of columns.
- A 2-D array **a**, which contains three rows and four columns can be shown as follows:

Eg: int a[3][4];

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

- Thus, every element in the array **a** is identified by an element name of the form **a[i][j]**, where 'a' is the name of the array, and 'i' and 'j' are the subscripts that uniquely identify each element in 'a'.

2. Initializing 2-D Arrays

2.1 Compile-Time Initialization of 2-D Arrays

- 2-D array can be initialized by specifying bracketed values for each row or as continuous elements.

```
int a[ 3 ][ 4 ] = {
        { 0 , 1 , 2 , 3 };
        { 4 , 5 , 6 , 7 },
        { 8 , 9 , 10 , 11 }
};
                (or)
int a[ 3 ][ 4 ] = { 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10 , 11 };
```

- Both the above initializations are equivalent, but this way is the best way to initialize the 2D array. It also increases the readability of the user.

2.2 Run-Time Initialization of 2-D Arrays

- To initialize the 2D array during run-time, the **nested loop structure** will be used;
  - outer for loop for the rows (first sub-script) and
  - the inner for loop for the columns (second sub-script) of the 2D array.

- To initialize the 2D array by using the run time initialization method :

```
for( i = 0 ; i < 3 ; i++)
   {
       for( j = 0 ; j < 3 ; j++)
       {
           scanf("%d", &ar1[ i ][ j ]);
       }
   }
```

3. Accessing 2-D Array Elements

- An element in a two-dimensional array is accessed by using the subscripts, i.e., row index and column index of the array.

```
int val = a[ 2 ][ 3 ];
```

- The above statement will take the 4th element from the 3rd row of the array.

Example Program

```
#include <stdio.h>
int main () {
 /* an array with 5 rows and 2 columns*/
  int a[ 5 ][ 2 ] = { { 0 , 0 } , { 1 , 2 } , { 2 , 4 } , { 3 , 6 } ,{ 4 , 8 }
};
  int i, j;

  /* to print each array element value */
  for ( i = 0; i < 5; i++ ) {

    for ( j = 0; j < 2; j++ ) {
      printf("a[ %d ] [ %d ] = %d\n", i, j, a [ i ][ j ] );
    }
  }
  return 0;
}
```

Output

```
a[ 0 ] [ 0 ]: 0
a[ 0 ] [ 1 ]: 0
a[ 1 ] [ 0 ]: 1
a[ 1 ] [ 1 ]: 2
a[ 2 ] [ 0 ]: 2
a[ 2 ] [ 1 ]: 4
```

a[3] [0]: 3
a[3] [1]: 6
a[4] [0]: 4
a[4] [1]: 8

Multi-dimensional Array

- The general form of a multidimensional array declaration is as follows:

  type name[size1] [size2] ... [sizeN];

- For example, the following declaration creates a three dimensional integer array

  int threedim[5] [10] [4];

Strings

- A String is an array of characters.
- They are terminated by a **null** character **'\0'**.
- The character array size is one more than the number of characters in the word.

Declaring and Initializing a String

- As string is a character array, it is declared and initialized as 1-D array.

char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};

or

char greeting[ ] = "Hello";

Example Program

```
#include <stdio.h>
int main () {
   char greeting[ 5 ] = {'H', 'e', 'l', 'l', 'o', '\0'};
   printf("Greeting message: %s\n", greeting );
   return 0;
}
```

Output

Greeting message: Hello

String Functions

- C offers several string related functions.
- These functions can be used to manipulate null terminated string.

Function	Purpose
strcpy(s1, s2);	Copies string s2 into string s1.
strcat(s1, s2);	Concatenates string s2 onto the end of string s1.
strlen(s1);	Returns the length of string s1.
strcmp(s1, s2);	Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
strchr(s1, ch);	Returns a pointer to the first occurrence of character ch in string s1.
strstr(s1, s2);	Returns a pointer to the first occurrence of string s2 in string s1.
strrchr(s1,ch)	Returns a pointer to the last occurrence of character ch in string s1.

Example Program

```
#include <stdio.h>
#include <string.h>
int main () {
   char str1[ 12 ] = "Hello";
   char str2[ 12 ] = "World";
   char str3[ 12 ];
   int  len ;

   /* copy str1 into str3 */
   strcpy(str3, str1);
   printf("strcpy( str3, str1) :  %s\n", str3 );

   /* concatenates str1 and str2 */
   strcat( str1, str2);
   printf("strcat( str1, str2):   %s\n", str1 );

   /* total length of str1 after concatenation */
   len = strlen(str1);
   printf("strlen(str1) :  %d\n", len );
   return 0;
```

```
}
```

Output

```
strcpy( str3, str1) :  Hello
strcat( str1, str2):   HelloWorld
strlen(str1) :  10
```

UNIT-III

Assignment-Cum-Tutorial Questions

Section-A

Objective Questions

1. **What is right way to initialize array?** **[ ]**

   A. int num[6] = { 2 , 4 , 12 , 5 , 45 , 5 };
   B. int n{ } = { 2 , 4 , 12 , 5 , 45 , 5 };
   C. int n{ 6 } = { 2 , 4 , 12 };
   D. int n(6) = { 2 , 4 , 12 , 5 , 45 , 5 };

2. **An array elements are always stored in _____ memory locations.**

3. **String concatenation means** **[ ]**
   A. Joins two strings.
   B. Extracting a substring out of a string
   C. Partitioning the string into two strings
   D. Comparing the two strings to define the larger one

4. **If the two strings are identical, then strcmp( ) function returns** **[ ]**

   A. 1 B. 0 C. -1 D. true

5. **The library function used to find the last occurrence of a character in a string is** **[ ]**
   A. laststr( ) B. strstr( ) C. strnstr( ) D. strrchr( )

6. **Which of the following function is more appropriate for reading in a multi-word string?** **[ ]**

   A. scanf( ) B. gets( ) C. printf( ) D. puts( )

7. **Below is an example of** **[ ]**
   int arr[5] [3] = { 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10 , 11 , 12 , 13 , 14 , 15 };

   A. 2-D Array B. 1-D Array C. 4-D Array D. 3-D Array

8. **Size of the array need not be specified, when** **[ ]**

   A. Initialization is run-time
   B. It is a declaration
   C. Both of the above
   D. None of the above

9. **What is the output of the following program?** **[ ]**

```
int main()
{
        int arr[5];
        arr[0] = 5;
        arr[2] = -10;
```

```
    arr[3/2] = 2;
    arr[3] = arr[0];
    printf("%d %d %d %d", arr[0], arr[1], arr[2], arr[3]);
    return 0;
}
```

a) 5 2 -10 5 b) 5 -10 5 2 c) -10 2 5 5 d) 5 5 2 -10

10. What will be printed after execution of the following code? [ ]

```
void main( ){
int arr[ 10 ] = { 1 , 2 , 3 , 4 , 5 };
 printf("%d", arr[ 5 ]);
}
```

A. Garbage Value B. 5 C. 6 D. 0 E. None of these

11. What will be the output of the program ? [ ]

```
#include<stdio.h>
void main()
{
   int a[5] = {5, 1, 15, 20, 25};
   int i, j, m;
   i = ++a[1];
   j = a[1]++;
    m = a[i++];
   printf("%d, %d, %d", i, j, m);
}
```

a) 3,2,15 b) 2,3 ,20 c) 2,1,15 d) 1,2, 5

12. What will be the output of the following program? [ ]

```
void main( ){
    char str1[ ] = "abcd";
    char str2[ ] = "abcd";
    if(str1==str2)
        printf("Equal");
    else
        printf("Unequal");
}
```

A. Equal B. Unequal C. Error D.None of these

13. What is the index number of the middle element of an array with 29 elements?
[ ]

A. 15 B. 14 C. 0 D.Programmer-defined’

14. What is the output of this C code? [ ]

```
#include<stdio.h>
#include<string.h>
void main( ){
   int a[ 2 ][ 3 ] = { 1, 2 , 3 , 4 , 5 };
   int i = 0, j = 0;
   for (i = 0; i < 2; i++)
   for (j = 0; j < 3; j++)
      printf("%d", a[ i ] [ j ]);
}
```

A. 1 2 3 4 5 0 B. 1 2 3 4 5 junk
C. 1 2 3 4 5 5 D. Run-Time Error

15. What will be the output of the program ? [ ]

```
#include<stdio.h>
#include<string.h>
void main( ){
   char str1[ 20 ] = "Hello", str2[ 20 ] = "  World";
   printf("%s", strcpy(str2, strcat(str1, str2)));
}
```

A. Hello World B. World C. WorldHello D.Hello

16. What will be the output of the program ? [ ]

```
#include<stdio.h>
void main( ){
   printf(5+"Good Morning");
}
```

A. Good Morning B. M C. Good D. Morning

17. What will be the output of the program ? [ ]

```
#include<stdio.h>
#include<string.h>
void main( ){
   char str[ ] = "Problem Solving\0Using C";
   printf("%s", str);
}
```

A. Problem Solving B. Problem Solving Using C
C. Using C D. None of these

18. [ ]

What is the output of the following program?

```
#include<stdio.h>
main()
{
int a[3] = {2,5,1};
printf("%d", a[a[0]]);
}
```

A. 0 B. 1 C. 2 D. Compile error

19. **[ ]**

What is the output of the following code

```
#include<stdio.h>
void main()
{
int a[3] = {5,8,2};
int b[3];
b = a;
printf("%d %d %d \n", a[0], a[1], a[2]);
printf("%d %d %d \n", b[0], b[1],b[2]);
}
```

A. 5 8 2 B. 5 8 2 C. 5 8 2 D. Compile error
5 8 2 0 0 0 5 0 0

Section-B

Descriptive Questions

1) Define array, and what is the advantage of using arrays?
2) Can we copy an array using the assignment operator? Justify your answer.
3) Draw flowchart for summation of n elements in an array.
4) What is a two-dimensional array in C? Illustrate different ways to initialize 2-D arrays.
5) How does an ordinary variable differ from an array?
6) Write the differences between the following functions:
   a. strcpy and strncpy
   b. strcat and strncat
   c. strcmp and strncmp

Programs:

1) Develop C code to compute mean and variance of a set of numbers.
2) Develop a C program to sort given set of elements in ascending order.
3) Develop a C program to find the Maximum and Minimum elements in 1-D array.
4) Develop a C program for addition of two matrices.
5) Develop a C program to compare two strings using string handling functions.
6) Develop a C program to copy one string to other without using string handling functions.

Section C:

1. what is the output printed if we execute following code void main()

```
{
        char a[] = "GATE2011";
        printf("%s", a + a[3] - a[1];

}
```

What will be the output of the above program? _____

GATE CS 2011

2. A program P reads in 500 integers in the range [0,100] representing the scores of 500 students. It then prints the frequency of each score above 50. what would be the best way for P to store the frequencies? [ ]

a)An array of 50 numbers.
b)An array of 100 numbers.
c)An array of 500 numbers
d) A dynamically allocated array of 550 numbers

GATE CS 2005

3. Consider the following declaration of a 'two-dimensional array in C: [ ]

```
char a[100][100];
```

Assuming that the main memory is byte-addressable and that the array is stored starting from memory address 0, the address of a[40][50] is
(a) 4040 (b) 4050 (c) 5040 (d) 5050

GATE CS 2002

4. An n*n matrix V is defined as follows [ ]
V[i,j]=i-j for all i,j, 1<=i<=n;1<=j<=n;
The sum of the elements of the array V is
(a) 0 (b) n-1 (c) n^2-3n+2 (d) n^2(n+1)/2

GATE CS 2000

5. Let A be a two-dimensional array declared as follows:

```
A: array [1 …. 10] [1 …… 15] of integer;
```

Assuming that each integer takes one memory location, the array is stored in row-major order and the first element of the array is stored at location 100, what is the address of the element A[i][j] [ ]
(a)15i +j + 84 (b) 15j + i + 84
(c) 10i + j + 89 (d) 10j + i + 89

GATE CS 1998

UNIT-4

Objective:

Familiarize about pointers and functions in C language.

Syllabus:

Pointers – Declaration, initialization and operations.
Functions – General form of functions, Passing parameters by value and by address, recursive functions, dynamic memory allocation functions, storage classes, pointers and arrays and String handling functions, Problem solving using functions.

Learning Outcomes:

At the end of the unit student will be able to:

- Understand the control pointers and functions in C.
- Solve moderate problems on computer using pointers and functions in C.

Learning Material

Functions:

Definition: A function is a self contained block of program statements that performs a particular task.

- We have written 'C' programs using three functions namely, main, printf and scanf.
- Every program must have a main function to indicate where the execution of the program begins and ends.
- If the program is large and if we write all the statements of that large program in main itself, the program may become too complex and large.
- As a result, the task of debugging, testing and even understanding will become very difficult.
- So, the best way to develop a large program is to construct it from smaller pieces (or) modules.
- A module can be a single function (or) a group of related functions carrying out a specific task.
- Usually, it is easier to break down a difficult task into a series of smaller tasks and then to solve those subtasks individually and later combined into a single unit.
- These subtasks are called User-defined functions.

Advantages:

- Using functions one can avoid rewriting the same code again and again.
- It is easy to write a function that does a particular job.
- It facilitates top-down modular programming. That means, the complexity of the entire program can be divided into simple subtasks and function can be written for each subtask.
- The length of the source program can be reduced.
- Saves memory space.
- Easier to write, testing and debugging individual functions.
- It increases program readability.

- A function can be shared by other programs.

General form of a function :

```
type function-name(argument list)   -----→ function header.
{
        Local variable declaration;
        Statement 1;
        Statement 2;                                    function body
        .........
        .........
        Return(expression);
}
```

- Here, type specifies the type of data that the function returns. A function may return any type of data except an array.
- The type and argument list are optional.
- An unspecified type is always assumed by the compiler to int. int is the default type when no type specifier is present.
- A function returns a type other than int, it must be explicitly declared.
- The function-name is any valid identifier.
- The argument list contains receive values from a calling function.
- We have two types of arguments:
  - Formal arguments
  - Actual arguments
- The Formal arguments are defined in the calling function.
- The data which is passed from the calling function to called function are called the Actual arguments. The actual arguments are passed to the called function through a function call.
- A function may (or) may not send a value back to the calling function. The value which is sent to the calling function is the return value of the function. It is achieved through the return statement.
- The return statement returns a value to the calling function. When the return statement is encountered the control is immediately passed back to the calling function.

Characteristics of functions :

- Any function can return only one value.
- Parameter argument list is optional.
- Return statement indicates exit from the function and return to the point from where the function was invoked.
- A function can call any number of items.
- A call to the function must end with a semicolon.
- Any 'C' function cannot be defined in other function.
- When a function is not returning any value, void type can be used as return type.
- 'C' allows recursion i.e..., a function can call itself.

Passing arguments to a function :

Arguments to a function are usually passed in two ways.

1. Call by value.
2. Call by reference.

Call by value : When a value is passed to a function via actual argument, the value of an actual argument is copied into the function. Therefore, the value of corresponding formal argument can be changed within the function, but the value of the actual argument within the calling function will not change.

Call by address: In this, the address of each argument is passed to the function. By this method, the changes made to the parameters of the function will affect the variables which called the function.

- **Return statement :**

The return statement serves two purposes:

- Transferring control back to the calling program.
- It returns the value present in the paranthesis after returns the calling program.

The general form of return statement is :

return; ←--- doesn't return any value.

(or)

return(expression); ←----- it returns a value of expression.

The return statement can be any of the following types :

```
return 0;
return (*x);
return (max);
return 'C';
return "true";
return sum;
```

Calling a function :

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

Syntax : fun-name(arg1,arg2,....);

Example :

```
void main( )
{
        int a, b, sum;
        printf("\n Enter values for a and b");
        scanf("%d%d",&a,&b);
        sum=add(a, b);
        printf("\n sum of two elements= %d",sum);
}
int add(int x, int y)
{
        int c;
        c=x+y;
        return c;
```

```
}
```

Categories of function :

According to the arguments and return values, there are four types of user defined functions.

1. Functions with no arguments and no return values.
2. Functions with arguments and no return values.
3. Functions with arguments and return values.
4. Functions with no arguments and return values.

1)Functions with no arguments and no return values :

In this type of function, the called function doesn't take arguments from the calling function and it will not give any return value to the calling function.

Syntax: → for function calling:

```
function name( );
```

→ for function definition:

```
void fun-name( )
{
    local variable declaration;
    statements;
    .....
    .....
}
```

Example:

```
void main( )
{
    char name[20];
    clrscr( );
    printf("Enter your name: ");
    scanf("%c", name);
    hai( );
}
void hai( )
{
    int time;
    printf("Enter time");
    scanf("%d",&time);
    if( time>=0 && time<=12)
        printf("Good morning");
    else if( time>12 && time<18)
        printf("Good afternoon");
    else if( time>=18 && time<=23)
        printf("Good evening");
    else
        printf("You entered an invalid time");
}
```

2) Functions with arguments and no return values :

In this type of functions, the called function can take the arguments from the calling function but it will return any values to the calling function.

Syntax : → for function definition:

```
void fun-name(datatype arg1, datatype arg2,.....)
{
    local variable declaration;
    statements;
    .....
    .....
}
```

→ For function calling:

```
fun-name(arg1, arg2, ..... );
```

Example:

```
/*swapping of two elements*/
#include<stdio.h>
void swap(int x, int y);
 void main( )
{
    int a,b;
    printf("\n Enter values of a and b");
    scanf("%d%d",&a,&b);
    printf("\n Before swapping the values are: a=%d \t b=%d",a,b);
    swap(a,b);
}
void swap(int x, int y)
{
    int t;
    t=x;
    x=y;
    y=t;
    printf("\n After swapping the values are : \n  x= %d  \t
    y=%d",x,y);
}
```

3) Functions with arguments and return values :

In this type of functions, the called function can take the arguments from the calling function and it will give some return values to the calling function.

Syntax: → for function definition:

```
return type fun-name(datatype arg1, datatype arg2,.....)
{
    local variable declaration;
    statements;
    .....
    .....
}
```

→ For function calling:

```
var-name= fun-name(arg1, arg2, ..... );
```

Example:

```
/*factorial of a given number*/
void main( )
{
        int res,n;
        clrscr( );
        printf("Enter any number for finding factorial");
        scanf("%d",&n);
        res=factorial(n);
        printf("The factorial of the %d is %d",n,res);
        getch( );
}
int factorial(int n)
{
        int i, fact=1;
        for(i=0; i<=n; i++)
                fact=fact * i;
        return fact;
}
```

4) Functions with no arguments and return values :

In this type of functions, the called function returns some value to the calling function. But it doesn't take any arguments from the calling function.

Syntax :

→ for function definition:

```
return type fun-name( )
{
        local variable declaration;
        statements;
        .....
        .....
        return(exp);
}
```

→ For function calling:

```
var-name= fun-name( );
```

Example :

```
/*factorial of a given number*/
void main( )
{
        int res;
        clrscr( );
        res=fact( );
        printf("The factorial is %d", res);
}
int fact( )
{
        int n,i, f=1;
        printf("\n Enter n value");
        scanf("%d",&n);
```

```
        for(i=0; i<=n; i++)
                f=f * i;
        return f;
}
```

Recursion:

A function, which invokes itself repeatedly until some condition is satisfied, is called a recursive function.

- The normal function will be called by main function whenever the function name is used.
- On the other hand, the recursive function will be called by itself repeatedly, until some specified condition has been satisfied.
- This technique is used to solve problems whose solution is expressed in termsof successively applying the same set of steps.

For example:

the factorial of a positive integer number 'n' is defined as

n!=n(n-1)(n-2).....1 with 1!=1, 0!=1.

The above formula can also be written as

n!=n(n-1)!

For example 5!=(5)(4!)
=5*4*3!
=5*4*3*2!
=5*4*3*2*1!
=5*4*3*2*1
= 120.

In 'C', a recursive function that calculates n! can be written as follows.

```
factorial(int n)
{
        if(n==1)
            return 1;
        else
            return(n*factorial(n-1));
}
```

/*C program for finding factorial of a given number without recursion*/

```
#include<stdio.h>
#include<conio.h>
int fact(int);
void main( )
{
        int n,f;
        clrscr( );
        printf("\n Enter n value");
        scanf("%d",&n);
        f=fact(n);
        printf("\n The factorial of %d is %d", n, f);
        getch( );
}
int fact(int x)
```

```
{
        int f;
        for(i=0; i<=x; i++)
                f=f * i;
        return f;
}
```

/*C program for finding factorial of a given number with recursion*/

```
#include<stdio.h>
#include<conio.h>
int fact(int);
void main( )
{
        int n,f;
        clrscr( );
        printf("\n Enter n value");
        scanf("%d",&n);
        f=fact(n);
        printf("\n The factorial of %d is %d", n, f);
        getch( );
}
int fact(int x)
{
        if(x==0)
            return 1;
        else if(x==1)
            return 1;
        else
            return(x*fact(x-1));
}
```

/*C program for generating Fibonacci sequence without recursion*/

```
#include<stdio.h>
#include<conio.h>
void fib(int);
void main( )
{
        int n;
        clrscr( );
        printf("\n Enter n value");
        scanf("%d",&n);
        fib(n);
        getch( );
}
void fib(int x)
{
        int i, first=0, second=1, next;
        for(i=0; i<n; i++)
        {
                if(n<=1)
```

```
                next=i;
                else
                {
                        next = first + second;
                        first = second;
                        second = next;
                }
                printf("%d \t",next);
        }
}
```

/*C program for generating Fibonacci sequence with recursion*/

```
#include<stdio.h>
#include<conio.h>
int fib(int);
void main( )
{
        int n, f;
        clrscr( );
        printf("\n Enter n value");
        scanf("%d",&n);
        for(i=1; i<=n; i++)
        {
                f=fib(i);
                printf("%d \t",f);
         }
        getch( );
}
int fib(int x)
{
        if(x==1)
             return 0;
        else if(x==2)
             return 1;
        else
             return fib(x-1) + fib(x-2);
}
```

Storage class:

Storage class in C decides the part of storage to allocate memory for a variable, it also determines the scope of a variable. All variables defined in a C program get some physical location in memory where variable's value is stored. Memory and CPU registers are types of memory locations where a variable's value can be stored. The storage class of a variable in C determines the life time of the variable if this is 'global' or 'local'. Along with the life time of a variable, storage class also determines variable's storage location (memory or registers), the scope (visibility level) of the variable, and the initial value of

the variable. There are four storage classes in C those are automatic, register, static, and external.

Types of Storage Classes:

There are four storage classes in C they are as follows

Automatic Storage Class

Register Storage Class

Static Storage Class

External Storage Class

1. Automatic Storage Class:

A variable defined within a function or block with auto specifier belongs to automatic storage class. All variables defined within a function or block by default belong to automatic storage class if no storage class is mentioned. Variables having automatic storage class are local to the block which they are defined in, and get destroyed on exit from the block.

The following C program demonstrates the visibility level of auto variables.

```
#include <stdio.h>
int main( )
{
 auto int i = 1
{
   auto int i = 2;
   {
    auto int i = 3;
    printf ( "\n%d ", i);
   }
   printf ( "%d ", i);
 }
 printf( "%d\n", i);
}
```

OUTPUT
3 2 1

2. Register Storage Class

The register specifier declares a variable of register storage class. Variables belonging to register storage class are local to the block which they are defined in, and get destroyed on exit from the block. A register declaration is equivalent to an auto

declaration, but hints that the declared variable will be accessed frequently; therefore they are placed in CPU registers, not in memory. Only a few variables are actually placed into registers, and only certain types are eligible; the restrictions are implementation-dependent. However, if a variable is declared register, the unary & (address of) operator may not be applied to it, explicitly or implicitly. Register variables are also given no initial value by the compiler.

```
#include<stdio.h>
int main()
{
int num1,num2;
register int sum;
printf("\nEnter the Number 1 : ");
scanf("%d",&num1);
printf("\nEnter the Number 2 : ");
scanf("%d",&num2);
sum = num1 + num2;
printf("\nSum of Numbers : %d",sum);
return(0);
}
```

3. Static Storage Class

The static specifier gives the declared variable static storage class. Static variables can be used within function or file.Unlike global variables, static variables are not visible outside their function or file, but they maintain their values between calls. The static specifier has different effects upon local and global variables.

visible only to the function or block in which it is defined. In simple terms, a static local variable is a local variable that retains its value between function calls. For example, the following program code defines static variable i at two places in two blocks inside function staticDemo(). Function staticDemo() is called twice within from main function. During second call static variables retain their old values and they are not initialized again in second call of staticDemo().

```
#include <stdio.h>

void staticDemo()
{
  static int i;
  {
    static int i = 1;
    printf("%d ", i);
    i++;
  }
  printf("%d\n", i);
  i++;
}

int main()
```

```
{
  staticDemo();
  staticDemo();
}
```

Output:

```
1 0
2 1
```

```
/* staticdemo.c */
#include <stdio.h>
static int gInt = 1;
static void staticDemo()
{
  static int i;
  printf("%d ", i);
  i++;
  printf("%d\n", gInt);
  gInt++;
}

int main()
{
  staticDemo();
  staticDemo();
}
```

Output:

```
0 1
1 2
```

Static variables have default initial value zero and initialized only once in their lifetime.

4. External Storage Class

The extern specifier gives the declared variable external storage class. The principal use of extern is to specify that a variable is declared with external linkage elsewhere in the program. To understand why this is important, it is necessary to understand the difference between a declaration and a definition. A declaration declares the name and type of a variable or function. A definition causes storage to be allocated for the variable or the body of the function to be defined. The same variable or function may have many declarations, but there can be only one definition for that variable or function.

When extern specifier is used with a variable declaration then no storage is allocated to that variable and it is assumed that the variable has already been defined elsewhere in the program. When we use extern specifier the variable cannot be initialized because with extern specifier variable is declared, not defined.

In the following sample C program if you remove extern int x; you will get an error "Undeclared identifier 'x'" because variable x is defined later than it has been used in printf. In this example, the extern specifier tells the compiler that variable x has already been defined and it is declared here for compiler's information.

```
#include <stdio.h>

extern int x;

int main()
{
  printf("x: %d\n", x);
}

int x = 10;
```

Pointers:

Pointer is one of the most powerful feature of C-language, which have number of advantages.

- Pointers enable us to access a variable that is defined outside the function.
- This is because the memory addresses are global to all functions, whereas local variable names are meaningful only within the function in which they are declared.
- Pointers reduce the length and complexity of a program. i.e.., pointers are used for saving memory.
- Pointers provide a way to return more than one value from a function.
- Pointers can be used to pass arrays and strings more conveniently from one function to another.
- Pointers increase the program execution speed.
- Pointers provide us dynamic memory allocations.

Definition of pointers:

- Pointer is a variable that holds the address of another variable.
- When we declare a variable, an appropriate memory location is allocated to the variable by the compiler to hold the value of a variable.
- This memory location will have its own address.
  Consider the statement, int i=10;

This statement tells the C compiler to perform the following actions.

1) Reserves a space in memory to hold the integer value.
2) Associate the name 'i' with this memory location.

i → location name
10 → value at location
65514→ location address

Now we may access the value '10' by using either the name 'i' or the address 65514.Since memory location addresses are simply numbers, they can be assigned to some variables which can be forced in memory, like another variable. Such variables that hold memory address are called pointers.

Declaration of pointer:

Syntax: datatype *variable name;

- Like another variables, pointers must be declared before they can be used.
- It is declared in a special way by preceding a '*' before the name of the variable.

- Datatype is the type of the pointer. It specifies the type of the object that the pointer can point to.
  Ex: int *x;
- The above statement states that 'x' is a pointer variable and is going to contain the address of a variable which is of integer type.
- We can read as " x is a pointer to int" (or) "x points to an object of type integer".
  Similarly, float *a;
- It declares 'a' as a pointer to a floating-point variable.

Initialization of pointers:

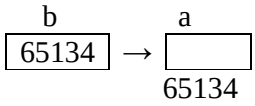
- Pointers should be initialized either when they are declared or in an assignment.
- A value cannot be assigned to a pointer variable directly as an ordinary variable.
- Instead it must be assigned a value of a variable which is already declared.

Syntax: datatype variable name1;

```
datatype *variable name = &variable name1;      /*Initialization at declaration time*/
                          (or)
        datatype *variable name;
        variable name = &variable name1;
```

Ex:
```
int a;
int *b=4a;
 (or)
int a;
int *b;
b=4a;
```

- In the above example, 'a' is an ordinary integer variable and 'b' is a pointer that points to an integer variable 'a'.
- The address of the variable 'a' is assigned to pointer 'b' by using &(address operator) that mean the variable 'b' holds the address of variable 'a'.
- A pointer may be initialized to '0' (zero), NULL or an address.
- A pointer with the value NULL points to nothing.
- Initializing a pointer '0' (zero) is equivalent to initializing a pointer to NULL, but NULL is preferred.



'b' is a pointer.

Pointer Operators:

- C contains two special operators : '&' and '*'.
- The '&' operator has been used to denote the address of the variable in the scanf function.
- This operator is known as address operator (or) Unary operator that returns the address of the variable it precedes and it is not restricted to the scanf function.
- It may precede only a variable name (or) array element, never a constant (or) expression.
  For example, int i=10;
  int *iptr;

iptr=&i; -----→ (1)

(1) statement assigns the address of 'i' to pointer variable 'iptr'. Variable 'iptr' is said to "points to i".

- Before a pointer is initialized, it should not be used.
- The '*'operator, commonly referred to as the indirection operator (or) dereferencing operator, returns the value stored at the address that it precedes.
  For example,
  printf("%d", *iptr);
- The above statement prints the value of the variable 'i', namely 10. Using '*' in this manner is called dereferencing a pointer.
- The '*' operator is also called "value at address" operator.
- Therefore '*iptr' and 'i' both represents the same data item [i.e.., the contents of the same memory cell].

Accessing a variable through its Pointer :

Once a pointer has been assigned the address of a variable, we can access the variable using the pointer by placing an indirection operator before the pointer.

Consider the following statements,

```
int a, *aptr, b;
aptr=&a;
a=5;
b=*aptr;
```

- The first line declares 'a' and 'b' as integer variables and 'p' as a pointer variable pointing to an integer.
- The second line assigns the address of 'a' to the pointer variable 'aptr'.
- The third line assigns the value '5' to 'a'.
- The fourth line contains the indirection operators.
- When the operator '*' is placed before a pointer variable in an expression(on the right hand side of the equal sign),the pointer returns the value of which the pointer value is address. In this case, *aptr returns the value of the variable 'a'. because aptr is the address of quantity.
- The '*' can be remembered as 'value at address'.
- Thus the value of 'n' would be '5'.

Example:

```
void main( )
{       int x,y;
        int *ptr;
        x=10;
        ptr=&x;
        y=*ptr;
        printf("value of x is %d",x);
        printf("%d is stored at address %u \n",x,&x);
        printf("%d is stored at address %u \n",*&x,&x);
        printf("%d is stored at address %u \n", *ptr, ptr);
        printf("%d is stored at address %u \n",y,& *ptr);
```

```
        printf("%d is stored at address %u \n",ptr,&ptr);
        printf("%d is stored at address %u \n",y,&y);
        *ptr=25;
        printf("\n now x=%d\n",x);
}
```

Output:

```
Value of x is 10
10 is stored at address 4104
10 is stored at address 4104
10 is stored at address 4104
10 is stored at address 4104
4104 is stored at address 4106
10 is stored at address 4108
Now x=25
```

Pointer expressions and pointer arithmetic;

In general, pointers may be used like other variables. So, pointers can be used in expressions.

For example, assuming the declarations.

```
int p=5;
int q=7;
int *xptr, *yptr,Z1,Z2;
```

The following statements are valid.

```
xptr=4p;
yptr=4q;
Z1=(*xptr) * ( *yptr);
*xptr=*xptr+3;
Z2=xptr-yptr;
*yptr=*yptr-1;
```

- In addition to the '*' and '&' operators, there are only four other arithmetic operators that may be applied to pointer variables +,++,- and --.
- A pointer may be incremented(++) or decremented(--), an integer may be added to a pointer(+ or +=), an integer may be subtracted from a pointer(- or -=), or one pointer may be subtracted from another pointer provided that they are pointing to the same array.
- Pointer arithmetic differs from "normal" arithmetic in the manner that it is performed relative to the base type (such as int, float, char etc..) of the pointer.
- Each time a pointer is incremented, it will point to the next item, as defined by its base type, beyond the one currently pointed to.
- For example, assume that an integer pointer called ptr contains the address 2000.
- After the statement, ptr++; Executes, ptr++ will have the value 2002, assuming integers are two bytes long.
- If the system used four bytes to store an integer, then ptr++ will have the value 2004.

- Similarly, if an integer pointer called 'y' contains the address 4104,then the statement

  y+=2;
- Would produce 4112 (4104+2*4) assuming an integer is stored in 4 bytes of memory.
- The statement y- =2, would set back to 4104.
- In general, incrementing a pointer using the ++ operator, increments the address it stores by a value equal to sizeof(type) where type is the data type of the variable pointed to

  (i.e.., 1 byte for char, 2 bytes for int, 4 bytes for float).
- We may not use pointers in multiplication (or) division , or may not take the modulus of a pointer and two pointers cannot be added.
- A pointer can be assigned to another pointer if both pointers are of the same type.
- Otherwise, a cast operator must be used to convert the pointer on the right of the assignment to the pointer type on the left of the assignment.
- We can apply the increment and decrement operators to either the points itself (or) the object to which it points to.
- For example, assume that 'i' point to an integer that contains the value'3'.
- The statement (*i)++; (note: parentheses are necessary.) will increment the value pointed by 'i'.

Ex:
```
main( )
{
    int *i,j;
    j=3;
    i=&j;
    printf("\n  Address of j=%u",i);
    (*i)++;
    Printf("\n   i=%u  j=%d",i,j);
}
```

Output:

Address of j=65524
I=65524 j=4

- In addition to arithmetic operations, pointers can be compared using relational operators.
- However, pointer comparisons only make sense of the pointer relate to each other, if they both point to the same object.

Multiple indirection:

- A pointer to a pointer is a form of multiple indirection a chain of pointers.
- Normally, a pointer contains the address of a variable.
- When we define a pointer to pointer, the first pointer contains the address of second pointer, which points to the location that contains the actual value as shown below.

Pointer ---------- > pointer ---------- > variable

Address

Value

- A variable that is a pointer to a pointer must be declared 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;

Ex:
```
void main( )
{
        int var;
        int *ptr, **pptr;
        var=3000;
        ptr=&var;
        pptr=&ptr;
        printf("\n value of var: %d",var);
        printf("\n value available at *ptr: %d", *ptr);
        printf("\n value available at **pptr: %d", **pptr);
}
```

Output:

Value of var: 3000

Value of variable at *ptr: 3000

Value available at **pptr: 3000

Void pointer:

- It is a special type of pointer.
- It can point any data type.
- The limitation is that the pointed data cannot be referenced directly by using '*' operator.
- Since its length is always undetermined.
- Therefore, typecasting must be used to turn the void pointer to pointer of datatype which we can refer.

Example:
```
void main( )
{
        int a=5;
        double b=3.48;
        void *vp;
        vp=4a;
        printf("\n a=%d", * ((int *)vp));
        vp=ab;
        printf("\n b=%d", * ((double*)vp));
}
```

NULL pointer:

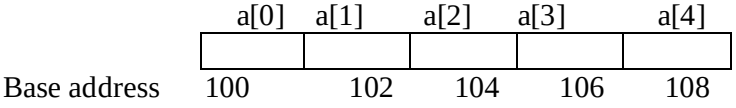
- A null pointer is a special pointer value that pointers nowhere.

- The most straight forward way to get a null pointer in the program is by using the predefined constant NULL, which is defined by several standard header files, including<stdio.h> , <stdlib.h> and <string.h> .
- To initialize a pointer to a null pointer, the following code can be used.
  #include<stdio.h>
  Int *ip=NULL;

Pointers and Arrays :

Pointers and one-dimensional arrays:

- In C language the elements of an array can be accessed through index.
- But 'C' also provides a special way of array handling through pointers.
- There is a very close relationship between arrays and pointers.
- An array name in 'C' is very much like a pointer but there is a difference between each other.
- The pointer is a variable that can appear on the left side of an assignment operator.
- The array name is a constant and cannot appear on the left side of an assignment operator.
- When an array is declared, the compiler allocates a base address and a block of contiguous memory locations to suit the number of elements.
- The first element of an array has the index.
- The array name without an index has special meaning in 'C'.
- It represents the address of the first element of an array.
- The base address is the location of the first element in the array.
- Compiler also defines the array name as a constant pointer to the first element.
- Suppose we declare an array as follows:
  int a[5]={1,2,3,4,5}
- The first elements of a will be stored as follows:



- We can also refer elements of an array as follows:
- *a or *(a+0) refers to the zeroeth element of the array i.e..., 1.
- Similarly, *(a+1) refers to the first element of the array i.e....,2 and so on.
- This is illustrated by the following program.

```
void main( )
{
        int i;
        int x[5]={1,2,3,4,5};
        for(i=0;i<n;i++)
        printf("\n   x[%d]=%d*(x+%d)=%d",i,x, *(x+i));
}
```

Output:

x[0]=1 *(x+0)=1

x[1]=2 *(x+1)=2
x[2]=3 *(x+2)=3
x[3]=4 *(x+3)=4
x[4]=5 *(x+4)=5

- Assume that integer pointer variable 'aptr' has been declared.
- Since an array name without an index is a pointer to the firs element of the array, we can set 'aptr' equal to the address of the first element in array 'a' with the statement

  aptr=a;
- This statement is equivalent to

  aptr=&a[0];
- Now, we can access every value of a using 'aptr++' to move from one element to another.
- The relationship between 'a' and 'aptr' is shown below.
  - aptr=&a[0]=100
  - aptr+1=&a[1]=102
  - aptr+2=&a[2]=104
  - aptr+3=&a[3]=106
  - aptr+4=&a[4]=108
- The array element "a[2]" can alternatively be referenced with pointer expression

  *(aptr+2)
- The '2' in the above expression is the offset to the pointer.

Example:

```
/*C program that calculates and print sum and average of 'n' given numbers using a pointer*/
#include<stdio.h>
void main( )
{
        int i,n;
        float *p, sum, avg, x[20];
        printf("\n Enter number of elements");
        scanf("%d",&n);
        printf("%f",&x[i]);
        p=x;
        sum=0.000000;

        for(i=0;i<n;i++)
        {
                sum+ = *p;
                p++;
        }
        avg=sum/(float)n;
        printf("\n Average=%f",avg);
        printf("\n Sum=%f",sum);
}
```

Pointers and two-dimensional arrays:

- A two dimensional array can be thought of as a collection of one-dimensional arrays, each indicating a row.

- Therefore, we can define a two-dimensional array as a pointer to a group of contiguous one-dimensional arrays.
- The declaration int a[2][3]; can be thought of as a collection of one dimensional array of 2 elements, each of which is a one dimensional array 3 elements long.
- Imagine 'a' to be a one-dimensional array and a[0] refers to first element and a[1] refers to second element.
- In two-dimensional array, a[0] gives the address of the zeroeth one-dimensional array and a[1] gives the address of first one-dimensional array.

int a[2][3]={11,12,13,14,15,16}

a[0][0]

	1000	1002	1003
A	0	1	2
0	11	12	13
1	14	15	16
	1006	1008	1010

a[1][0]

- The name of the array points to the starting address of the array.
- The element in row1,column2 can be accessed by writing a[1][2].
- Similarly a[1][2] is interpreted as (a[1]+2) would give the address (1006+2).
- Obviously (1006+2) would give the address 1010. The value of a[1][2] is given by * (a[1]+2).
- But a[1] is same as * (a+1). Therefore, value of a[1][2] is * (* (a+1)+2).
- Thus, a[i] points to the i^{th} row of the array and a[i]+j points to the j^{th} element in the i^{th} row of an array.
- The subscript 'j' actually acts as an offset to the base address of the i^{th} row.
- Thus, the following expressions refer to the same element.

a[i][j]
* (a[i]+j)
* (* (a+i)+j)

Example :

```
/* program for addition of two matrices using pointers */
#include<stdio.h>
void main( )
{
        int a[3][3], b[3][3], c[3][3];
        int *aptr, *bptr, *cptr,i;
        aptr=&a[0][0];
        bptr=&b[0][0];
        cptr=&c[0][0];
        printf("\n Elements of matrix A");
        for(i=1; i<10; i++)
        {
                scanf("%d",aptr);
                aptr++;
        }
        printf("\n Elements of matrix B");
```

```
        for(i=1; i<10; i++)
        {
                scanf("%d",bptr);
                bptr++;
        }
        printf("Matrix c \n");
        aptr=&a[0][0];
        bptr=&b[0][0];
        for(i=1; i<10; i++)
        {
                if ((i==4) !! (i==7)) printf("\n");
                *cptr=(*aptr)+( *bptr);
                Printf("%d", *cptr);
                aptr++;
                bptr++;
                cptr++;
        }
}
```

Pointers and functions :

The arguments can be passed to a function in two ways.

1. Call by value (sending the values of arguments)
2. Call by reference (sending the address of the argument)

1) call by value :

In this method, the value of each of the actual argument in the calling function is copied into the corresponding formal arguments of the called function.

- With this method the alteration of values of formal argument will not effect the actual arguments which passed their values to parameters.
- The usual mode of function call in 'C' is call by value. This means in general you cannot alter the actual arguments. But, you can alter through 'call by reference'.

Consider an example program of swapping of two elements.

```
/*swapping of two elements*/
#include<stdio.h>
void swap(int x, int y);
void main( )
{
        int a,b;
        printf("\n Enter values of a and b");
        scanf("%d%d",&a,&b);
        printf("\n Before swapping the values are: a=%d \t b=%d",a,b);
        swap(a,b);
}
void swap(int x, int y)
{
        int t;
        t=x;
```

```
        x=y;
        y=t;
        printf("\n After swapping the values are : \n  x= %d  \t  y=%d",x,y);
}
```

- When a, b arguments are passed to the function, only a copy of its value is passed to the formal parameters 'x' and 'y'.
- Any changes made to the formal parameters within the function does not effect the value of 'a' and 'b'.

Consider a=5,b=6.

a	b
5	6
100	104

- When passing these two values by using call by value, the formal parameters 'x' and 'y' holds the values '5' and '6'.

x	Y
5	6
110	108

- After swapping the values are of 'x' and 'y' are

x	Y
6	5
110	108

- But ,this modification does not effect to 'a' and 'b'. That means 'a' and 'b' holds the values '5' and '6' only after swapping.

a	b
5	6
100	104

2) <u>call by reference :</u>

In this method, the address of actual arguments are passed to the called function.

- When arguments are passed by reference the address of actual arguments in the calling function are copied into formal arguments of the function.
- The contents of these addresses can be accessed freely and hence we would be able to manipulate them.
- Moreover, any change that is made to the contents of the address will be recognized in both calling function and called function.
- While we pass address to the function, the formal arguments must be pointers.

Consider an example program of swapping of two elements.

```
/*swapping of two elements*/
#include<stdio.h>
void swap(int *, int *);
void main( )
{
        int a,b;
        printf("\n Enter values of a and b");
        scanf("%d%d",&a,&b);
        printf("\n Before swapping the values are: a=%d \t b=%d",a,b);
```

```
        swap(&a,&b);
        printf("\n After swapping the values are: a=%d \t b=%d",a,b);
}
void swap(int *x, int *y)
{
        int t;
        t=*x;
        *x=*y;
        *y=t;
}
```

- In the above program, address of 'a' and 'b' are passed to the called function.

Let us assume a=5,b=6.
Address of 'a' is 102 and 'b' is 104.

a	b
5	6
102	104

- Now, the formal parameters 'x' and 'y' holds the address 'a' and 'b'.

x	Y
102	104
100	106

- After swapping the values are of 'a' and 'b' are

a	b
6	5
102	104

- Since, 'x' and 'y' points to 'a' and 'b' we are indirectly accessing 'a' and 'b' variables only. So, the changes in formal parameters will effect the actual parameters.

Passing arrays to functions:

There is no restriction in passing any number of values to a function. Like the values of simple variables, it is also possible to pass the values of the array to a function. To pass an array to a called function, it is sufficient to list the name of the array without any brackets and the size of the array as arguments.

For example, the call to the function small (x,n) will pass all the elements contained in the array 'x' of size 'n'.
The "small" function header will look like:

```
small( float x[20], int n)
```

The function "small" is defined to take two arguments, the array name and the size of the array to specify the number of elements in the array.

Remember that when an array is passed to a fuction as an argument, only the address of the first element of the array is passed, but not the actual values of the array elements. The functions uses this address for manipulating the array elements.

Syntax :

```
Type function-name(type1 array1, type2 array2,.....,type1 arg1,.....)
{
        Statements;
```

```
        }
```

Example :

```
/* Mean, S.D, variance */
#include<stdio.h>
#include<conio.h>
void main( )
{
        float a[20],n,sd;
        int i;
        clrscr( );
        printf("\n Enter array size");
        scanf("%d",&n);
        printf("\n Enter the %d values");
        for(i=0; i<n; i++)
                scanf("%f",&a[i]);
        sd=std-dev(a,n);
        printf("\n Standard deviation=%f",sd);
        getch( );
}
float sd-dev(float a[ ],float n)
{
        int i;
        float sum=0,x;
        x=mean(a,n);
        printf("\n Mean=%f",x);
        for(i=0; i<n; i++);
                sum+ = (pow(x-a[i]),2);
        return(sqrt(sum/n));
}
float mean(float a[ ], float n)
{
        float sum=0;
        int i;
        for(i=0; i<n; i++)
                sum+ =a[i];
        return(sum/n);
}
```

/* Mean, S.D, variance using functions */

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
float mean(float [ ]);
float variance(float, float [ ]);
float std-dev(float);
void main( )
{
        float a[30], sd, m, v, n;
```

```
        int i;
        clrscr( );
        printf("\n Enter array size");
        scanf("%f",&n);
        printf("\n Enter the %d values");
        for(i=0; i<n; i++)
                scanf("%f",&a[i]);
        m= mean(a, n);
        v= variance(m, a, n);
        sd= std-dev(v);
printf("\t Mean= %f    \t Variance= %0.2f    \t Standard deviation= %0.2f",m, v, sd);
        getch( );
}
float mean(float a[ ],float n)
{
        int i;
        float sum= 0.0;
        for(i=0; i<n; i++)
                sum+ =a[i];
        return(sum/n);
}
float variance(float m, float a[ ], float n)
{
        float ssum= 0.0;
        int i;
        for(i=0; i<n; i++)
                sum+ = pow((m-a[i]),2);
        return(sum/n);
}
float std-dev(float v)
{
        return(sqrt(v));
}
```

Dynamic memory allocations:

The process of allocating memory at run time is known as "Dynamic memory allocations".

- The exact size of array is unknown until the compile time i.e..., time when a compiler compiles code written in a programming language into a executable form.
- The size of an array you have declared initially can be sometimes insufficient and sometimes more than required.
- Dynamic memory allocation allows a program to obtain more memory space while running (or) to release space when no space is required.
- Memory management functions are used for allocating and freeing memory during execution of a program.
- The memory management functions are defined in "stdlib.h" header file.

Memory allocation process:

- Global variables and program instructions are stored in permanent storage area.
- Local variables are stored in stack.
- The memory space between these two regions is known as Heap.
  The region is used for dynamic memory allocation.

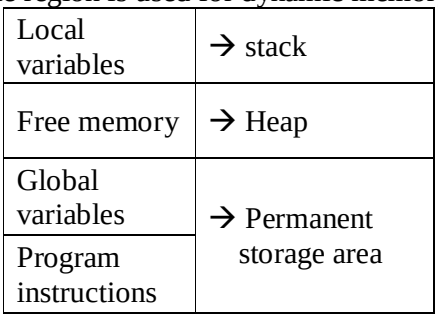


Fig: Storage of a 'C' program.

Memory management functions :

- malloc( )
- calloc( )
- free( )
- realloc ( )

malloc( ) :

The name malloc stands for "memory allocation". The function malloc( ) reserves a block of memory of specified size and return of type void which can be casted into pointer of any form.

Function prototype is :

```
void *malloc(size-t  size);
```

syntax for malloc :

```
ptr = (cast-type * ) malloc(byte-size);
```

Here, "ptr" is pointer of cast-type. The malloc function returns a pointer to an area of memory with size of byte-size. If the space is insufficient, allocation fails and returns NULL pointer.

Example:

```
Ptr = (int *)malloc(100 * sizeof(int));
```

This statement will allocate 200 bytes (since the size of int is 2 bytes) and the pointer points to the address of first byte of memory.

calloc( ) :

The name calloc stands for "contiguos allocation". The only difference between malloc( ) and calloc( ) is that, malloc( ) allocates single block memory whereas calloc( ) allocates multiple blocks of memory each of same size and sets all bytes to zero.

Function prototype is :

```
void *calloc(size-t  count, size-t eltsize);
```

syntax for calloc :

```
ptr = (cast-type * ) calloc(n, element-size);
```

This statement will allocate contiguous space in memory for an array 'n' elements . If the space is insufficient, allocation fails and returns NULL pointer.

Example:

```
ptr = (float *)calloc(25, sizeof(float));
```

This statement allocates contiguous space in memory for an array of 25 elements each of size float i.e., 4 bytes.

→ calloc( ) function is normally usec for derived datatypes i.e..., Arrays, Structures.

free( ) :

Dynamically allocated memory with either malloc( ) or calloc( ) does not get return on its own. The programmer must use free( ) explicitly to release space.

Function prototype is :

```
void free(void * block);
```

syntax for free( ):

```
free(ptr);
```

This statement cause the space in memory pointer by "ptr" to be deallocated.

realloc( ) :

If the previously allocated memory by using malloc( ) or calloc( ) is insufficient or more than sufficient. Then, you can change memory size previously allocated using realloc( ).

Function prototype is :

```
void *realloc(void *block, size-t size);
```

syntax for realloc( ) :

```
ptr = realloc(ptr, newsize);
```

Here, "ptr" is allocated with "newsize".

Example:

```
#include<stdio.h>
#include<stdlib.h>
void main( )
{
    int  *ptr, i, n1, n2;
    printf("Enter size of array");
    scanf("%d",&n1);
    ptr=(int *)malloc(n1*sizeof(int));
    printf("\n Address of previously allocated memory: ");
    for(i=0; i<n1; i++)
        printf("%u\n",ptr+i);
    printf("\n Enter new size of an array: ");
    scanf("%d",&n2);
    ptr=realloc(ptr,n2);
    for(i=0; i<n2; i++)
        printf("%u \n",ptr+i)
}
```

Example program by using calloc( ) :

```
/*C program to find sum of 'n' elements.To perform this program,allocate memory dynamically using calloc function*/
#include<stdio.h>
#include<stdlib.h>
void main( )
{
    int  *ptr, i, n, 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("\n Enter elements of array");
        for(i=0; i<n; i++)
        {
                scanf("%d",ptr+i);
                sum+ =* (ptr+i);
        }
        printf("sum=%d",sum);
        free(ptr);
}
```

Example program by using malloc( ) :

/*find sum of 'n' elements.To perform this program,allocate memory dynamically using malloc function*/

```
#include<stdio.h>
#include<stdlib.h>
void main( )
{
        int  *ptr, i, n, sum=0;
        printf("Enter number of elements");
        scanf("%d",&n);
        ptr=(int *)malloc(n* sizeof(int));
        if(ptr==NULL)
        {
                printf("Error! Memory not allocated");
                exit(0);
        }
        printf("\n Enter elements of array");
        for(i=0; i<n; i++)
        {
                scanf("%d",ptr+i);
                sum+ =* (ptr+i);
        }
        printf("sum=%d",sum);
        free(ptr);
}
```

UNIT-IV

Assignment-Cum-Tutorial Questions

Section-A

Objective Questions

1. A pointer is [ ]
   a) A keyword used to create variables
   b) A variable that stores address of an instruction
   c) A variable that stores address of other variable
   d) All of the above

2. Integer pointer points to a memory block of size: [ ]
   a) 1 byte b)2 bytes c) 4 bytes d) none of the above

3. Using _____ we can allocate memory block which is further splitted in to sub blocks and clears their contents to zero?
   [ ]
   a) malloc( ) b) calloc( ) c) realloc( ) d) freealloc( )

4. Which of the following pointer expressions are valid [ ]
   i. ptr+10
   ii. ptr*10
   a) only i is valid b) both i and ii c)only ii d) none

5. _____ operator is used to get the address and _____ to get value at address stored in a pointer variable. [ ]
   a) &,* b) (address of),(value at) c) (type),- d) ^,&

6. The keyword used to transfer control from a function back to the calling function is
a) switch b) goto c) go back d) gofrom [ ]

7. What is the size of a void pointer ? [ ]
a) 0 b) 1 c) 2 d) NULL

8.What is the output of this C code? [ ]

```
#include <stdio.h>
void main()
{
char *s = "hello";
char *p = s;
printf("%p\t%p", p, s);
}
```

a) Different address is printed b) Same address is printed
c) Run time error d) Nothing

9.What is the output of this C code?

```
#include <stdio.h>
void main()
{
   char *s= "hello";
   char *p = s;
   printf("%c\t%c", p[0], s[1]);
}
```

a) Run time error b) h h c) h e d) h l

10. Functions can return any type of values [ ]
a) True b) False

11. If a function contains two return statements successively, the compiler will generate warnings. Yes/No ? [ ]
a) Yes b) No

12. Usually recursive programs demand more memory when compared to programs with non-recursive functions. [ ]
a) True b) False

Section -B

Descriptive Questions

1. Define Function. Explain the Categories of Functions with an example for each.
2. Differentiate pre-defined and user defined functions.
3. Distinguish between parameter passing by value, parameter passing by address with suitable example.
4. What is recursion? Explain with an example?
5. What is Pointer? How Operations can be performed on Pointers?

6. Implement the Dynamic Memory Allocation concept with an example program
7. Illustrate the Chain of pointers with example program.
8. Explain about array of pointers with an example?

Programs:

9. Write the c program to print sum of all elements of the array using pointers.
10. Write the c program for swapping of two numbers using call-by -value and call-by-reference.
11. Write a c program to print factorial of given number using recursion.
12. Write a c program to print GCD of two number using recursion.
13. Write a function to exchange two numbers with and without using temporary variable.
14. Write a recursive program for generating n^{th} number in the fibonacci series.
15. Write a C program to sort a given set of numbers in ascending order using functions

SECTION-C

Gate Questions

1. Consider the following function implemented in C

[GATE-2017]

```
void printxy (int x, int y) {
int *ptr ;
x = 0;
ptr = &x;
y = * ptr;
* ptr = l;
print f ("%d, %d," x, y);
}
```

The output of invoking printxy (l, l) is [ ]

(A) 0,0 (B) 0,1 (C) 1,0 (D) 1,1

2.Consider the following program: **Gate(2016)**

```
int f(int *p, int n)
{
if (n <= 1)
        return 0;
else
        return max(f(p+1,n-1),p[0]-p[1]);
}
int main()
{
        int a[] = {3,5,2,6,4};
        printf("%d", f(a,5));
}
int max(int a, int b)
{
```

```
    if(a>b)
      return a;
    else
      return b;
 }
```

The value printed by this program is_____

3. What will be the output of the following C program?` [ ]

Gate(2016)

```
void count(int n)
{
    static int d=1;
    printf("%d ", n);
    printf("%d ", d);
    d++;
    if(n>1) count(n-1);
    printf("%d ", d);
}
void main()
{
    count(3);
}
```

(A) 3 1 2 2 1 3 4 4 4
(B) 3 1 2 1 1 1 2 2 2
(C) 3 1 2 2 1 3 4
(D) 3 1 2 1 1 1 2

4. Point out the compile time error in the program given below

```
#include<stdio.h>
int main()
{
   int *x;
   *x=100;
   return 0;
}
```

a) Error: invalid assignment for x
b) Error: suspicious pointer conversion
c) No Error
d) None Of the Above

5.What will be the output of the following pseudo-code when parameters are passed by reference and dynamic scoping is assumed? [ ]

Gate(2016)

```
int a=3;

void n(x)
```

```
{
        x = x * a;
        printf("%d",x);
}
void m(y)
 {
        a = 1;
        a = y - a;
        n(a);
        printf("%d",a);
}
void main()
{
        m(a);
}
```

(A) 6, 2

(B) 6, 6

(C) 4, 2

(D) 4, 4

6. What does the following fragment of C-program print?

```
void main()
{
        char c[] = "GATE2011";
        char *p =c;
        printf("%s", p + p[3] - p[1]);
        return 0;
}
```

[ ]

(a) 4 b) 2 c) 8 d) Garbage Value

7. Consider the following code [ ]

```
void get (int n)
{
        if (n < 1) return;
        get (n–1);
        get (n–3);
        printf ("%d", n);
}
```

If get(6) function is being called in main( ) then how many times will the get() function be invoked before returning to the main( )?

(a) 15 (b) 25 (c) 35 (d) 45

8. The value printed by the following program is _____ [ ]

```
void f(int* p, int m)
{
```

```
        m = m + 5;
        *p = *p + m;
        return;
}
void main()
{
        int i=5, j=10;
        f(&i, j);
        printf("%d", i+j);
}
```

(a) 30 (b) 40 (c) 25 (d) 35

9. Consider the following C program segment. [ ]

```
#include <stdio.h>
int main( )
{
  char s1[7] = "1234", *p;
  p = s1 + 2;
  *p = '0' ;
  printf ("%s", s1);
}
```

What will be printed by the program?

(a) 12 (b) 120400 (c) 1204 (d) 1034

10. What does the following program print?

```
#include<stdio.h>
void f(int *p, int *q)
{
 p = q;
*p = 2;
 }
int i = 0, j = 1;
int main()
{
 f(&i, &j);
 printf("%d %d \n", i, j);
 getchar();
 return 0;
}
```

UNIT-V

Objective:

Familiarize about structures, unions and bit fields in C language.

Syllabus:

Structures -Definition, declaration, initialization, accessing structure members, nested structures, array of structures, structures and functions, unions.

Problem solving- Implement a structure to read and display the Name, Date of Birth and Salary of Employees, Functions to perform read, add and write two complex numbers using Structures.

Learning Outcomes:

- At the end of the unit student will be able to
- understand the structures, unions and bit fields in C.
- solve moderate problems on computer using structures, unions and bit fields in C.

STRUCTURES AND UNIONS:

➢ **Structure:**
- A structure is a collection of different data types that are logically grouped together and referenced under a single name.
- The main use of structure is to represent the different attributes or characteristics of an entity.
- It creates a format, which may be used to declare many other variables in that format.
- It is a user defined data type.

➢ **Syntax of a Structure Definition:**

```
struct  structure_name / tag name
{
    datatype   var_name, var_name1,.....;
    datatype   var_name,var_name1,.....;
    ..........
       ..........
} ;
```

❖ The keyword struct tells the compiler that a structure type is being defined.

❖ structure_name (or) tag_name serves as a name that may be used for a particular template of the structure.

- The data type need not be the same. Different variable types can be separated by a semicolon.
- The elements of a structure are also referred to as fields (or) members and are enclosed within curly braces.
- The closing brace is followed by a semicolon.

For example, a structure may contain student's roll-number, student name, average marks and any other relevant information about the student. Hence, student's data might be represented by the following individual variables.

```
int   roll_no;
char   name[20];
int   m1,m2,m3,total;
float   avg;
```

These variables can be grouped together into a single structure, using the following definition.

```
struct student
{
int    roll_no;
char   name[20];
int    m1,m2,m3,total;
float   avg;
};
```

This construction is often called a structure definition. It simply describes a format called template to represent information. The structure definition creates a new data type that is used to declare variables.

The allocation of memory takes place only when the structure variables are declared. A structure variable is similar to other variables.

➢ Structure variable declaration:

Just as any other variables, structure variables can also be declared using the tag name. Once you have defined a structure type, you can create variables of that type using the general form.

```
struct   tag-name   var-list;
```

For example,

```
struct student s1,s2,s3;
```

Here s1, s2 and s3 are variables of type 'struct student'.

Each one of these variables has seven members i.e.., roll no, name, m1, m2, m3, tot and avg.

It is also allowed to combine the definition of the structure type and structure variables declaration in one statement.

For example:

```
struct   student                    struct   student
{                                   {
```

```
    int    roll_no;
    char   name[20];
    int    m1, m2, m3, tot;    [or]
    float avg;
    }s1, s2, s3;
```

```
    int    roll_no;
    char   name[20];
    int    m1, m2, m3,tot;    float avg;
    };
    struct   student s1, s2,s3;
```

- **Initializing structure variable:**

The members of a structure variable can also be initialized in the same way as any other data types.

The general form of structure Initialization is

struct tag-name variable = {value1, value2,....value n};

The first value in the list is assigned to the first member; the second value in the list is assigned to the second member and so on.

Example:

```
struct student
{
        int roll_no;
        char name[20];
        int m1, m2, m3, total;
        float avg;
};
main( )
{
        struct student s1={101,"sai",25,30,39};
        ........
        ........
}
```

- **Accessing structure members:**

To access members of the structure, we have to use a dot(.) operator. A member of a structure can be accessed using the structure variable name and the member name separated by the dot(.) operator.

The dot(.) operator establish a link between the structure variable and member.

Syntax: structure_variable. member_name

- The dot operator is also known as structure member operator.
- The student's roll number of the structure s1 can be accessed by writing s1.roll_no.
- Similarly, the student's name of the structure s2 can be accessed by writing s2.name.

- **Assigning values to members:**

We can assign values to the members of a structure in the following ways:

```
strcpy ( s1.name, "Siri");
s1.roll_no=101;
```

We can also use scanf( ) or gets( ) to give values.

```
scanf ("%s", s1.name);      or gets(s1.name);
scanf("%d", &s1.roll_no);
```

- **Operations on Structure variables:-**

Structure Variables in assignment Statements:

One structure can be assigned to another, only when they are of the same structure type using the assignment operator.

Example: student1=student2;

The value of each member of student2 is assigned to the corresponding member of student1.

→ ***Comparison of structure variables:***

Two variables of the same structure type can be compared like other variables.

Examples: if student1 and student2 are of the same structure type, the statement

1. student1==student2 Compares all members of student1 and student2 and return1 if they are equal, otherwise 0.
2. student1 != student2 It returns 1 if all the members are not equal, otherwise 0.

- **Nested structure:-**

When a structure is placed within another structure as its member, then the resultant structure is called nested structure. This means one structure is defined within another structure.

Example:-

```
struct account
{
    int acc no;
      char name[30];
      float balance;
};
struct deposit
{
      float amount;
      struct account ac;
} customer;
```

[or]

```
struct deposit
  {
      float amount;
      struct amount
          {
          int acc_no;
              char
name[30];
          float balance;
        }ac;
        }customer;
```

Accessing members of nested structures:-

If a structure member is itself a structure, then a member of the embedded structure can be accessed by chaining all the concerned structure variables (from outer most to inner most) with the (.) dot operator.

To access the members contained in the inner structure, we would write.

```
customer.ac.acc_no;
customer.ac.name;
```

However, nesting can be up to any level. We can nest a structure, within another structure, which is in still another structure and so on.....

Ex:-

```
struct loan
{
        struct deposit
        {
                struct account
{
int acc_no;
char name[20];
float balance;
                }ac;
                float amount;
                float years;
        }dep;
        float amount;
        char date[10];
}loan 1;
```

- **Array of structures:-**

A similar type of structures placed under a common variable is called array of structures. The array will have individual structures as its elements.

For example, to store the details of 40 students in a class, it would be required to declare 40 different structure variables from s1 to s40, which is inconvenient. A better method would be to use an array of structures, each element of the array representing a structure variable.

An array of structures can be declared in two ways.

```
struct student                          struct student
{                                   {
    int roll no;                            int roll no;
```

```
        char name[20];                          char name[20];
        int m1, m2, m3, tot;        [or]        int m1, m2, m3, tot;
        float avg;                              float avg;
};                                          }s[40];
struct student s[40];
```

In the above example, s is a 40 element array of structures of the type student. All elements of the array s[40] are stored in adjacent memory locations.

Accessing elements in Array of structures:-

Any member of the structure can be accessed using the index number along with the structure name.

For example:

- To access the details of the student 10, we can use the statement
  s[9] since the subscript begins with 0.
- To access the name of 10th student, we write
  s[9].name;
- Similarly, if we want to print the marks of the first student. we can use the printf statement
  printf("%d %d %d", s[0].m1, s[0].m2, s[0].m3);

Initializing Array of structures:-

We can initialize an array of structures in the same way as a single structure.

```
struct student
{
int  roll no;
char  name[20];
int   m1, m2, m3, tot;
float  avg;
};
struct student s[2]={  {101, "abc", 36, 38, 39},
                       {102, "def", 37, 25, 40}
 };
```

Here, 's' is an array of 2 elements of type student. Thus, s[0] will be assigned to the first set of values, s[1] will be assigned the second set of values.

The C compiler will assign the values to its elements to the particular structure as given below.

For the first record

```
s[0].roll_no=101;
s[0].name="abc";
```

```
        s[0].m1=36;
        s[0].m2=38;
        s[0].m3=39;
   For the second record
     s[1].roll_no=102;
        s[1].name="def";
        s[1].m1=37;
        s[1].m2=25;
        s[1].m3=40;
```

Example:-

```
#include<stdio.h>
#include<conio.h>
struct student
{
        int    roll_no;
        char   name[20];
        int    m1, m2, m3,tot;
}s[10];
void main( )
{
        int n, i;
        clrscr( );
        printf("Enter no. of students :");
        scanf("%d", &n);
        for(i=0; i<n; i++)
        {
             printf("Enter details of student %d", i+1);
             scanf("%d %s %d %d %d", &s[i].roll_no, s[i].name, &s[i].m1,
                          &s[i].m2, &s[i].m3);
        }
        for(i=0; i<n; i++)
        s[i].tot=s[i].m1+s[i].m2+s[i].m3;
        printf("\n Details of %d students:",n);
        for(i=0; i<n; i++)
        {
      printf("%d \t %s \t %d \t %d \t %d \t %d \t"
                ,s[i].roll_no,s[i].name,s[i].m1,s[i].m2,s[i].m3,s[i].tot);
              printf("\n");
        }
        getch( );
```

```
}
```

- **Passing structures to Functions:-**

A function is a very useful aid to break a complex program into separate smaller parts or modules. Each such module is called a function and is needed to construct the complex program into a very simple one. Finally, we invoke all the different functions in the main program to get the complete solution of the complicated program.

'C' supports passing of structures as parameters to functions. There are three different ways to pass the values of a structure to (or) from a function.

Passing individual structure members:-

Structure members can be passed individually to functions, like ordinary variables and a single member can be returned via the return statement.

Example:-

```
#include<stdio.h>
#include<conio.h>
struct student
{
    int   roll_no;
    char   name[20];
};
void main( )
{
struct   student   s={101, "abc"};
clrscr( );
show(s.roll_no,s.name);
getch( );
}
void show(int  roll_no , char  name[ ])
{
    printf("%d \t %s", roll_no,name);
}
```

Passing a copy of the entire structure:-

Like an ordinary variable, a structure variable (i.e., an entire structure) can be passed as a parameter to the called function. Any changes to structure members within the function are not reflected in the original structure. A structure is passed in this manner will be passed by value.

Example:-

```
#include<stdio.h>
#include<conio.h>
struct   student
{
    int   roll_no;
    char   name[20];
};
void main( )
{
struct student s={101, “abc”};
clrscr( );
show(s);
getch( );
}
void show(struct   student   s1)
{
    printf(“%d \t %s”, s1.roll_no, s1.name);
}
```

Passing a copy of an entire structure by reference:-

A complete structure can also be passed as an argument to a function by passing a structured type pointer as an argument. In this concept, the address location of the structure is passed to the called function. A structure passed in this manner will be passed by reference rather than structure.

Example:-

```
#include<stdio.h>
#include<conio.h>
struct student
{
    int roll_no;
    char name[20];
};
void main( )
{
struct student s={101, “abc”};
struct student *s1=&s;
clrscr( );
show(&s);
getch( );
}
void show(struct student *s1)
{
```

```
        printf("%d \t %s", s1→roll_no, s1→name);
    }
```

Comparision between Array and Structure:-

	ARRAY		STRUCTURE
1)	Stores homogenous data. (same data types)	1)	Stores heterogeneous data. (different data types)
2)	Two arrays of same type cannot be assigned to one another directly.	2)	Structures are of same type can be assigned.
3)	Arrays can be initialized.	3)	Structures can also be initialized.
4)	Array is a combination of elements.	4)	Structure is a combination of members.
5)	No operators are required to access elements of an array.	5)	Operators like '.' or '→' are required to access members of a structure.

Unions:-

A union is a derived data type like a structure whose members share the same storage space within the computer's memory.

- The members of a union can be of any type.
- The number of bytes used to store a union must be at least enough to hold the largest number.

Since, the union stores values of different types in a single location it can handle only one member and thus one data type at any one time.

Declaration of union;-

The syntax for union is identified to that for a structure, except that the keyword struct is replaced with the keyword union.
The general form of a union is

```
union tag_name
{
        datatype  member-1;
        datatype  member-2;
        ……
        ……
}variable1,variable2,……..variable-n;
```

Example:-

```
union sample
{
        char c;
```

```
    int num;
    float f_num;
        }example;
```

This declares a variable example of type union sample with three members each with a different data type.

The fact that only one location is allocated for a union variable, the compiler allocates a piece of storage thet is large enough to hold the largest variable type in the union.

In the above union declaration, the members f-num requires 4 bytes which is largest among the members.

```
1001  1002   1003   1004
|< ch>|
|←--num--→|
|←------------f_num---------→|
```

Accessing union members:-

Union members can be accessed in the same manner as structure members using the dot (.) operator.

The individual members of example are referred to as

```
example.ch
example.num
example.f_num
```

If the variable of union is pointer then the members can be accessed by using '→' operator.

- Only one member of a union can be accessed at a time.
- Only that member which is last written, can be read. At this point other variables will contain garbage values.

Example:-

```
#include<stdio.h>
#include<conio.h>
union date
{
    char  name[20];
    int   day, month, year;
}person;
void main( )
{
clrscr( );
printf("\n Enter name: ");
scanf("%s", person.name);
printf(\n Enter day, month and year");
scanf("%d%d%d", &person.day, &person.month, &person.year);
```

```
        printf("Name and Date of birth");
printf("\n %s \t %d - %d - %d", person.name, person.day,
         person.month, person.year);
        getch( );
            }
```

Output:-

```
 Enter name: gopi
 Enter day, month and year12   2    2000
Name and Date of birth
 ╨pi     2000 - 2000 - 2000
```

Operations on union:-

The operations that can be performed on a union are the following:

- ✓ A union variable can be assigned to another union of same type.
- ✓ A union variable can be passed to a function as a parameter.
- ✓ The address of the union variable can be extracted by using the address operator (&).

DIFFERENCES BETWEEN STRUCTURE AND UNION:

	STRUCTURE		**UNION**
1)	*Definition* :- Collection of elements of different data types having common tag-name.	1)	*Definition* :- Collection of elements of different data types having common tag-name.
2)	struct sample { int i; char c; float f; };	2)	union sample { int i; char c; float f; };
3)	Syntax: struct tag-name { type 1 member 1; type 2 member 2; .... type n member n; };	3)	Syntax: union tag-name { type 1 member 1; type 2 member 2; .... type n member n; };
4)	Structure variable declaration:- struct sample s;	4)	Union variable declaration:- union sample u;
5)	Syntax: struct tag-name v1, v2,	5)	Syntax: union tag-name v1, v2,

	……v-n;		……v-n;
6)	Size of structure: 2+1+4=7 bytes	6)	Size of union: Largest of (2,1,4) = 4 bytes
7)	The memory size of the structure is the sum of the sizes of its member's data types.	7)	The memory size of the union is largest of the sizes of its member's data types.
8)	Structure stores different data types of data in different locations. i.e..., each member within a structure is assigned its own unique storage.	8)	union stores different data types of data in single memory location. i.e..., the member that composes a union all shares the same storage area.
9)	Initialization: struct sample s={10, 'A', 8.7}	9)	Initialization: s.i= 10; s.c= 'A'; s.c= 8.7;
10)	We can access all the members of structures simultaneously.	10)	We can access only one member at a time. We can't access all at a time.
11)	Distinct memory allocations are possible.	11)	Memory sharing is possible in union.
12)	Occupies more memory when compared to union.	12)	Occupies less memory when compared to structure.

13)

```
/*Example for structure*/
#include<stdio.h>
struct sample
{
    int i;
    char c;
    float f;
};
void main( )
{
    struct sample s={10, 'A', 8.7};
    printf("\n room=%d \n section=%c \n
                per=%f", s.i, s.c, s.f);
    printf("\n size of structure=%d bytes",
                sizeof(s));
}
```

13)

```
/*Example for union*/
#include<stdio.h>
union sample
{
    int i;
    char c;
    float f;
};
void main( )
{
    union sample u;
      u.i=10;
    printf("\n room=%d", u.i);
      u.c= 'A';
    printf("\n section=%c", u.c);
      u.f=8.7;
    printf("\n per=%f", u.f);
    printf("\n size of union=%d bytes",

sizeof(u));
}
```

- **User defined datatypes:**

Datatypes		
Primary	**Derived**	**User defined**
→ Int	→ array	→ typedef
→ char	→ structure	→ enum
→ float	→ union	

- **typedef:-**

'typedef' is a key word that allows the programmer to create new data type name for an existing data type.

→ No new data type is produced but an alternate name is given to a known data type.

General form of declaring typedef is:-

typedef existing data type new data type;

Example:-

If we want to declare an integer variable, we can use "roll_no" instead of "int".

```
typedef int roll_no;
```

→ Structure can also use typedef keyword.

Consider a structure

```
struct student
{
        int roll_no;
        char name[20];
};
struct student s1;
                ↖ structure variable
```

If we place typedef before struct in structure definition

```
typedef struct student
{
        int roll_no;
        char name[20];
}stud;
```

☜ ----- a new name to type "struct student".

Here 'stud' is not structure variable. It is structure type, that means we can declare a variable for the student structure by using the below statement.

```
stud s1;
```

Instead of

```
struct student s1;
```

- **Enumeration type:- (enum)**

An enumeration is a user-defined data type consists of integral constants and each integral constant is given a name.

The general form of declaring enumeration type:

enum tagname{member 1, member 2,.....member n}variable 1, variable 2,.....variable n;

[or]

```
enum tagname
{
        member 1,
member 2,
.....,
.....,
member n
} variable 1, variable 2,.....variable n;
```

The general form of declaring a variable to the enumeration type:

enum tagname variable 1, variable 2,.....variable n;

- The members are integer constants.
- By default, the first member is given the value 0, the second member is given the value 1 and so on.
- Members within the braces may be initialized. In this case, the next member given the value one more than the preceding number.

Example:-

```
enum day{mon, tue, wed, thur, fri, sat};
enum day day 1, day 2, day 3;
day1=mon;
day2=tue;
day3=wed;
```

Here day 1, day 2, day 3 are variables for the enum type 'day'.

- **Array within a structure:-**

C allows structures to contain arrays. So far, the members of structure have been declared as an ordinary data type such as int, char, float only. C permits members of a structure as an array data type also. We have already used an array of characters inside a structure. Similarly, we can use one or two dimensional arrays of type int or float.

Example:

```
struct student
{
int roll_no;
char name[20];
int sub[3];
}s[10];
```

Here, the member 'sub' contains 3 elements sub[0], sub[1] and sub[2] to hold the marks obtained by a student in three subjects.

For example:

s[2].sub[0]

refers to the marks scored in the first subject by the third student.

UNIT-V

Assignment-Cum-Tutorial Questions
Section-A

Objective Questions

1. A structure can be defined inside an union YES/NO [ ]

2. Bitfields are possible in both structures and unions [T/F]

3. What is the keyword used for declaring structure [ ]
A. Structure B. User-defined C. struct_tag D. struct

4. Which of the following are themselves a collection of different data types? [ ]

A. string **B.** structure **C.** char **D.** All of the mentioned

5. Which operator connects the structure name to its member name? [ ]

A. - **B.** dot . **C.** Both (b) and (c) **D**. None

6. Which of the following cannot be a structure member? [ ]

A. Another structure **B**. Function **C.** Array **D.** None

7. Number of bytes in memory taken by the below structure is? [ ]

```
struct test
{
        int    k;
        char  c;
};
```

A. Multiple of integer size **B**. integer size + character size

C. Depends on the platform **D**. Multiple of word size

8. A structure can be nested inside another structure. [T/F]

9. Size of a union is determined by size of the. [ ]

A. First member in the union **B.** Last member in the union

C. Biggest member in the union **D**. Sum of the sizes of members

10. Members of a union are accessed as_____ [ ]

A. union_name. member **B.** union--member

C. Both a & b **D.** None of the mentioned

11.A union cannot be nested in a structure.T/F [ ]

12.Which of the following data types are accepted while declaring bit-fields? [ ]

A. char **B.** float **C.** double **D.** none of the mentioned

13. The elements of union are always accessed using & operator [True/False]

Section-B

Subjective Questions

1. Define structure. Describe structure initialization with an example.
2. Point out the differences between structure and array.
3. How array of structures are defined? Give an example.
4. Define nested structure. How it differs from array of structures?
5. How to copy one structure to another structure of same data type, give an example.
6. Define Union? Differentiate structure and union?
7. Describe the significance of user-defined datatypes?.

Programs:

1. Write a C program to implement a structure to read and display the Name, date of Birth and salary of n number of employees.
2. Write a C program to implement a union to read and display the Name, date of Birth and marks of a student.
3. Write a C program to display the Name, Marks in three subjects and total marks of given number of students. (Using array of structures).
4. Write a C program which uses functions to perform the following operations using Structure:
   i) Reading a complex number ii) Writing a complex number
5. Write a C program which uses functions to perform the following operations using Structure:
   i) Addition of two complex numbers ii)Multiplication of complex number
6. Write a C program that illustrates the accessing and initializing members of a Union.
7. Write a C program to illustrate passing structure members to functions.

8. Write a C program to illustrate passing entire structure to function.

SECTION-C

Gate Questions

1.The following C declarations**(GATE2000)** [ ]

```
A struct node
{
        int i;
       float j;
};
struct node *s[10] ;
define s to be
```

A. An array, each element of which is a pointer to a structure of type node

B.A structure of 2 fields, each field being a pointer to an array of 10 elements

C.A structure of 3 fields: an integer, a float, and an array of 10 elements

D. An array, each element of which is a structure of type node.

2. Consider the following C declaration [ ]

```
struct {
        short s[5];
        union {
                float y;
                long z;
              }u;
        } t;
```

3.Assume that objects of the type short, float and long occupy 2 bytes, 4 bytes and 8 bytes, respectively. The memory requirement for variable t, ignoring alignment considerations, is **(GATE CS 2000)**

[ ]

A. 22 bytes **B**.14 bytes **C**. 18 bytes **D.**10 bytes

4. Which of the following structure declaration will throw an error?

A. struct temp{}s; [ ]
main(){}

B. struct temp{};
struct temp s;
main(){}

C. struct temp s;
struct temp{};
main(){}
D. None of the mentioned

5.What is the output of the following code**[GATE 2003]**

```
void main()
{
struct student
{
int no;
char name[20];
};
struct student s;
s.no = 8;
printf("%d", s.no);
}
```

A. Nothing **B.** Compile time error **C.** Junk **D.** 8

6. For what minimum value of x in a 32-bit Linux OS would make the size of s equal to 8bytes? [ ]

```
struct temp
{
int a  :13;
int b  :8;
int c  :x;
}s;
```

A. 4 **B.** 8 **C.** 12 **D.** 32.

UNIT-VI

Objective: Familiarize about File Handling Functions.

Syllabus: File Handling- Text and binary files, file handling functions, file processing operations – inserting, deleting, searching and updating a record and displaying file contents, random access to files.

Problem solving – Copy the contents of one file to another, count the number of characters, words and lines in a file

Learning Outcomes:

At the end of the unit student will be able to

- Understand file handling functions and file processing operations
- Solve moderate problems on computer using file processing operations

Files

Introduction

- When a large volume of data is involved, supplying data through the keyboard during the execution or displaying the output on the screen is not convenient.
- The input data can be stored on disks and the program may access the data from disks for processing.
- Similarly, the results may be stored on disks. For such applications, files are needed.

Definition:

A file is a place on the disk where a group of related data is stored. In C, file manipulations may be done in two ways:

- Low-level I/O using system calls
- High-level I/O using functions from standard I/O library

- The files accessed through the library functions are called Stream Oriented files and the files accessed with system calls are known as System Oriented files.

Streams and Files

- Streams facilitate a way to create a level of abstraction between the program and an input/output device. This allows a common method of sending and receiving data amongst the various types of devices available. There are two types of streams: text and binary.
- Text streams are composed of a set of lines. Each line has zero or more characters and is terminated by a new line character. Text streams consist of printable characters, the tab character, and the new-line character.
- Conversions may occur on text streams during input and output.
- Spaces cannot appear before a newline character, a text stream removes these spaces even though implementation defines it.
- A text stream, on some systems, may be able to handle lines of up to 254 characters long (including the terminating new line character).
- Binary streams are composed of only 0's and 1's. It is simply a long series of 0's and 1's. More generally, there need not be a one-to-one mapping between characters in the original file and the characters read from or written to a text stream.
- in case of the binary stream there will be one-to-one mapping because no conversion exists, and all characters will be transferred as such.

When a program begins, there are three available streams:

- Standard input (stdin) is the stream where a program gets its input data
- Standard output (stdout) is the stream where a program writes its output data.
- Standard error (stderr) is another output stream typically used by programs to output error messages.

File Operations

- Files are associated with streams and must be open in order to use it. The point of I/O within a file is determined by the file position.
- When a file is opened, the file position points to the beginning of the file unless the file is opened for an append operation - in which case the position points to the end of the file. The file position indicates where the next operation (read/write) will occur.
- When a file is closed, no more actions can be taken on it until it is opened again. Exiting from the main function causes all open files to be closed.

In C, 'FILE' is a structure that holds the description of a file and is defined in stdio.h.

Basic File operations are:

- Opening a File
- Reading from and/or writing into a File
- Closing the File

The logic is, the code must:

- define a local 'pointer' of type FILE (called file pointer)
- 'open' the file and associate it with the file pointer via fopen()
- perform the I/O operations using file I/O functions (ex. fscanf() and fprintf()) disconnect the file from the task using fclose()

General form:

```
FILE *fp;
fp = fopen("name", "mode");
fscanf(fp,format string",variablelist);
fprintf(fp, "format string", variable list);
fclose(fp );
```

- 'fp' is a file pointer or file handler.

- The 'name' is to represent filename and it is a string of characters. (Extensions can be specified like test.c, details.dat etc)

- The 'mode' argument in the fopen() specifies, the purpose/positioning of opening the file. It is a string enclosed within double quotes.

The 'mode' can be any of the following:

r read text mode
w write text mode (truncates file to zero length if it already exits or creates new file)
a append text mode for writing (opens or creates file and sets file pointer to the end-of-file)
rb read binary mode
wb write binary mode (truncates file to zero length if it already exits or creates new file)
ab append binary mode for writing (opens or creates file and sets file pointer to the end-of-file)
r+ read and write text mode
w+ read and write text mode (truncates file to zero length if it already exists or creates new file)
a+ read and write text mode (opens or creates file and sets file pointer to the end-of-file)
r+b/rb+ read and write binary mode
w+b/wb+ read and write binary mode (truncates file to zero length if it already exists or creates new file)
a+b/ab+ read and write binary mode (opens or creates file and sets file pointer to the end-of-file)

- If the file does not exist and it is opened with read mode (r), the file open fails and it will return NULL to file pointer.
- If the file is opened with append mode (a), all write operations occur at the end of the file regardless of the current file position.
- If the file is opened in the update mode (+), output cannot be directly followed by input and input cannot be directly followed by output without an intervening fseek(),

fsetpos(), rewind(), or fflush().

- fopen() returns the file pointer position for successful open and returns NULL, if the file does not open or the file does not exist.

- fclose() returns zero for successful close and returns EOF (end of file) when error is encountered in closing a file. By default, all the files opened are closed when the program is terminated.

- It is good to close all the files opened with fopen(), because files can be reopened only if they are closed.

- The Standard I/O provides variety of functions to handle files. It supports the following ways of reading from and writing into file:

  - Chaacter I/O
  - String I/O
  - Formatted I/O
  - Block I/O
  - Integer I/O

Character I/O

Using character I/O, one character (byte) can be written to or read from a file at a time.

Writing in to a file

- To write into a file, the file must be opened in 'w' mode The function putc() is used to write a byte to a file.

General Form:

putc(ch,fptr);

- This function writes the character ch into a file pointed by the file pointer fptr. This

fptr may be stdout, which represents standard output device, monitor as a file. On success, the character is returned.

- If an error occurs, the error indicator for the stream is set and EOF is returned.

Example: Program to create a text file (character file)

```
void main()

{
        FILE *fp; char c;
        if ((fp=fopen("sample.dat","w")) !=NULL)
        {
                while ((c=getchar()) != EOF) putc(c,fp);
                fclose(fp);
        }
        else
                printf("Error in opening a file");
}
```

Reading from a file

- The function getc() is used to read a byte from a file. This may be a macro version of fgetc.

General Form:

ch =getc (fptr);

- This function reads a character from the file and it is returned to the program defined character variable. After reading a character, the pointer is moved to the next position.
- The fptr may be stdin, which represents a standard input device, keyboard as a file. On success, the character is returned. If the end-of-file is encountered, EOF is returned

and the end-of-file indicator is set.

- If an error occurs, the error indicator for the stream is set and EOF is returned. The EOF is end of file status flag, which is true if end of file is reached, otherwise false.

Example : Program to read a character data from a text file

```
void main()
{
    FILE *fp; char c;
    if ((fp=fopen("sample.dat","r")) !=NULL)
    {
        while ((c=getc(fp)) != EOF)
            putchar(c);
        fclose(fp);
    }
    else
        printf("Error in opening a file");
}
```

String I/O

Using string I/O, string can be written to, or read from, a file at a time.

Writing a string in to a file

- The function used is fputs(). Writes a string to the specified stream till the last character is read but does not include the null character. On success, a nonnegative value is returned. On error, EOF is returned.

General Form:

fputs (str, fptr);

Reading a string from a file

- The function used is fgets(). Reads a line from the specified stream and stores it into the string pointed to by str. It stops when (n-1) characters are read, the newline character is read, or the end-of-file is reached, whichever comes first. The newline character is copied to the string. A null character is appended to the end of the string.
- On success, a pointer to the string is returned.
- On error, a null pointer is returned.
- If the end-of-file occurs before any characters have been read, the string remains unchanged.

General Form:

fgets(str,n,fptr);

Numeric I/O

Using numeric I/O, integers can be written to, or read from, a file at a time.

Writing integer in to a file

- The function used is putw(). This function writes an integer to a file. On success, a nonnegative value is returned. On error, EOF is returned.

General Form:

putw (i, fptr);

Reading integer from a file

- The function used is getw(). Reads an integer from the file and assigns it to the program defined numeric variable at the LHS.

General Form:

i = getw(fptr);

Formatted I/O

The formatted I/O functions can handle a group of data in a single call.
Writing formatted data to a file

- The function fprintf() is used. This function will write the values stored in the variables into a file pointed by fptr, according to the format specifier specified in format string.
- On success, the number of characters printed is returned.
- If an error occurred, -1 is returned.

General Form:

fprintf (fptr, format-string, variable-list);

- The fprintf() function takes the format string specified by the format argument and applies each following argument to the format specifiers in the string, in a left to right fashion.
- Each character in the format string is copied to the stream except for conversion characters which specify a format specifier.

Reading formatted data from the file

- The function used is fscanf().This function will read the formatted data from the file pointed by fptr, as specified by the format specifiers in format-string and stores in the variables, whose addresses are given in addresses-list.
- Reading an input field (designated with a conversion specifier) ends when an incompatible character is met, or the width field is satisfied.
- On success, the number of input fields converted and stored is returned.
- If an input failure occurs, EOF is returned.

General Form:

fscanf(fptr, format-string, addresses-list);

- The fscanf() function takes input in a manner that is specified by the format argument and stores each input field into the corresponding arguments, in a left to right fashion.
- Each input field is specified in the format string with a conversion specifier which specifies how the input is to be stored in the appropriate variable.
- Other characters in the format string specify characters that must be matched from the input, but are not stored in any of the following arguments. If the input does not match, the function stops scanning and returns.
- A white space character may match with any white space character such as space, tab, carriage return, new line, vertical tab, or form feed, or the next incompatible character.

Block I/O

Block I/O is used to read or write a specified number of bytes. The data handled by block input/output function will be in 'raw data format' (i.e. bytes of data).

Writing in to a file

- The function used is fwrite().Transfers a specified number of bytes beginning at a specified location in memory to a file.

- Used to write a structure or an array of structures to an output file.
- The function writes data from the array pointed to by ptr to the given stream. It writes 'n' blocks of size 'size'. The total number of bytes written is (size*n).
- On success the number of elements written is returned.
- On error the total number of elements successfully written (which may be zero) is returned.

General Form

fwrite (ptr, size, n, fp);

- ptr pointer to the data block (source)
- size size of each block (number of bytes to be written) n number of blocks to be written
- fp file pointer (destination)

Reading from a file

- The function used is fread(). Reads data from the given stream into the variable pointed to by ptr. It reads 'n' number of elements of size 'size'.
- The total number of bytes read is (size*n).
- On success the number of elements read is returned.
- On error or end-of-file, the total number of elements successfully read (which may be zero) is returned.

General Form

fread (&str, size, n, fp);

- &str destination memory address
- size size of each block (number of bytes to be read) n number of blocks to be read
- fp file pointer (source)

Example: Program using Block I/O

```
void main()
{
        FILE *fptr;
         struct tag
        {
              char name[10];
              int age ;
        }stud[10] , stud1[10];
        int i ;
        clrscr();
        fptr=fopen("ex.dat" , "w" );
        for(i=0 ; i<5 ; i++)
          scanf("%s %d ",stud[i].name , &stud[i].age); fwrite(&stud ,sizeof(stud[0]) , 5 , fptr);
        fclose(fptr);
        fptr = fopen("ex.dat" , "r" );
        fread(&stud1 , sizeof(stud1[0]) , 5 , fptr);
         printf(" \n\n printing the values "); for(i=0 ; i<5 ; i++)
        printf("\n %s \t %d " , stud1[i].name , stud1[i].age);

}
```

Random Access to Files

- The functions discussed earlier are to be used for reading and writing data sequentially.
- In some applications, it may be necessary to access some part of the file directly.
- This can be achieved by using the functions fseek(), ftell() and rewind().

ftell()

- This function takes a file pointer and returns a long int, which corresponds to the current file pointer position.
- If it is a binary stream, then the value is the number of bytes from the beginning of the file.
- If it is a text stream, then the value is a value usable by the fseek() function to return the file position to the current position.
- On success, the current file position is returned.
- On error, the value -1L is returned and error number (errno) is set.

General Form:

n = ftell(fptr);

fseek()

- This function sets the file position to the given offset (specified in long integer format).

General Form:

fseek(fptr, offset, from_where)

- The argument offset signifies the number of bytes to seek from the given 'from_where' position. The argument from_where can be:

SEEK_SET Seeks from the beginning of the file 0

SEEK_CUR	Seeks from the current position of File pointer	1
SEEK_END	Seeks from end of the file	2

- On a text stream, from_where should be SEEK_SET and offset should be either zero or a value returned from ftell().
- The end-of-file indicator is reset. The error indicator is NOT reset.
- On success, zero is returned.
- On error, a nonzero value is returned.

Example

fseek (fp, 0L, 0);	Move the file pointer to the beginning.
fseek (fp, 0L, 2);	Move the file pointer to the end of file.
fseek (fp, 10L, 0);	Move after 10 bytes from the beginning.
fseek (fp, 10L, 1);	Move after 10 bytes from the current
fseek (fp, -10L, 1);	Move backward 10 bytes from the current
fseek (fp, -10L, 2);	Move backward 10 bytes from the EOF.

rewind()

- This function sets the file position to the beginning of the file of the given stream. The error and end-of-file indicators are reset.

General Form:

rewind(fptr);

Program illustrating fread():

```
#include<stdio.h>

struct st

{
 int id;
 char name[20];

}s;

void main()

{
 FILE *fp;
 clrscr();

 fp=fopen("bin.txt","rb");

 fread(&s,sizeof(s),1,fp);

 printf("%d%s",s.id,s.name);

 getch();

 fclose(fp);

}
```

Program illustrating fwrite():

```
void main()

{
 struct tag
 {

    char name[20];
```

```
    int no;
 }v;
 FILE *fp;
 fp=fopen("input.txt","wb");
 printf("enter name and numer");
 scanf("%s%d",v.name,&v.no);
 fwrite(&v,sizeof(v),1,fp);
 fclose(fp);
}
```

Program illustrating fprintf():

```
#include<stdio.h>
void main()
{
 FILE *fp;
 fp=fopen("input.txt","w+");
 fprintf(fp,"hellooooooooo world");
 fclose(fp);
}
```

UNIT-VI

Assignment-Cum-Tutorial Questions
Section-A

Objective Questions

1. A file is a collection of records, that are related logically [True/False] [ ]

2. Expand EOF [ ]

A. End Of File B. Exit Of File C. Error in operating File D. None

3. FILE data type defined in stdio.h allows us to define a file pointer. [ ]

A. false B. true C .no valid answer

4. What is the operating mode in which file can be read as well as written [ ]

A. “a” B. “r+ C. “w+” D. both b&c

5. SEEK_SET signify that the offset is relative to current position in the file when we define the function fseek() [ ]

A. false B. true C. no valid answer

6. Which of the following opens a file? [ ]

A. fscanf B. open C. fopen D. create( )

7. What does file mode "wb" signify? [ ]

A. Open a text file for writing B. Open a binary file for appending data

C. Create a binary file for writing D .Create a text file for writing

8. fread() and fwrite() functions are used to handle records in a file [T/F] [ ]

9. Which of the following can read input from a file? [ ]

A.fscanf B.fgets C.fread D.all the above

10. In fclose(fp), fp is the _____ of the file that needs to be closed.

11. In a file containing the line "I am a boy\r\n" then on reading this line into the array str using fgets(). What will str contain? [ ]

A."I am a boy\r\n\0" B."I am a boy\r\0"

C."I am a boy\n\0" D."I am a boy"

12. What is the function prototype of fwrite()? [ ]

A.size_t fwrite(size_t sz, size_t n, File *fp, const void *ptr)

B.size_t fwrite(const void *ptr, size_t sz, size_t n, File *fp)

C.size_t fwrite(File *fp, const void *ptr, size_t sz, size_t n)

D.size_t fwrite(size_t sz, const void *ptr, size_t n, File *fp)

13. Which of the following indicates end of file? [ ]

A.fscanf B.ferror C.feof D.all of the options

14. If *fp is the file pointer, long int fseek(FILE *fp) is the prototype of function fseek().

[ ]

A.false B.true C.no valid answer

15. What does fp point to in the program ? [ ]

```
#include<stdio.h>
int main( )
{
        FILE *fp;
        fp=fopen("trial", "r");
       return 0;
  }
```

A. The first character in the file

B. A structure which contains a char pointer which points to the first character of a file.

C. The name of the file.

D. The last character in the file.

16. What does fp point to in the program ? [ ]

```
#include<stdio.h>
int main()
{
   FILE *fp;
   fp=fopen("trial", "r");
   return 0;
}
```

A. The first character in the file

B. A structure which contains a char pointer which points to the first character of a file.

C. The name of the file.

D. The last character in the file.

17. Match the following

I	II
A.SEEK_SET	i. 0
B.SEEK_CUR	ii. 2

C.SEEK_END iii. 1

18. To print out a and b given below, which of the following printf() statement will you use? [ ]

```
float a=3.14;
double b=3.14;
```

A.printf("%f %lf", a, b);

B.printf("%Lf %f", a, b);

C.printf("%Lf %Lf", a, b);

D.printf("%f %Lf", a, b);

19. Which files will get closed through the fclose() in the following program? [ ]

```
#include<stdio.h>
int main()
{
FILE *fs, *ft, *fp;
fp = fopen("A.C", "r");
fs = fopen("B.C", "r");
ft = fopen("C.C", "r");
fclose(fp, fs, ft);
return 0;
}
```

A. "A.C" "B.C" "C.C"

B. "B.C" "C.C"

C. "A.C"

D. Error in fclose()

Section-B

Subjective Questions

1. What is meant by Random access to files? Explain fseek() ftell() in detail.

2. Write a C-Program to create a file and Also explain the file operations used in this program

3. Write a c program to create separate files for even and odd numbers in an existing file
4. Explain different types of files in detail
5. Write a c program to merge contents of 2 different files into a single file
6. Explain how fgets() and fputs() work with an example program
7. Explain usage of putw and getw with an example program
8. What is the difference between fscanf( ) and fgets( )?

Programs:

9. Write a c program that uses any 3 file handling functions
10. Write a c program for searching for a record in a file
11. Write a c program illustrating the usage of feof() and ferror()
12. Write a c program for billing checkout counter of a super market
13. Write a c program for preparing consolidated attendance/ marks sheet
14. Write a c program for illustrating usage of fprintf() and fscanf()
15. Write a c program for illustrating fseek() and explain the arguments of fseek()
16. Write a c program for performing string handling operations on file contents.

C. Questions asked in Competative exams

1. Write a program that reads two command line arguments. The first one is a string and the second one a file name. The program then should search the file, printing all lines that contain the string. Use fgets( ) rather than get( ).

   Hint: use the function strstr( ).

2. Find the error(s) in the following program segment and rewrite the correct program **[GATE-2007]**

```
int main ( )
{
        int * fp;
        int, k;
        fp = fopen ("pizza");
        for (k = 0; k < 30; k++)
                printf ("Jill likes pizza.\n", fp);
        fclose ("pizza");
        return 0;
}
```