

1 byte = 8 bits

Kalpna Coaching Classes

Thane / Nerul / Dadar / Vileparle / Borivali

Gargi Learning Centre

Dombivli

Bubble Sort:

```

for (i=0; i < n-1; i++)
{
    for (j=0; j < n-1-i; j++)
    {
        if (arr[j] > arr[j+1])
        {
            temp = arr[j];
            arr[j] = arr[j+1];
            arr[j+1] = temp;
        }
    }
}

```

SPA

SEM - II

Notes By

Selection Sort:

```

for (i=0; i < n-1; i++)
{
    min = i;
    for (j=i+1; j < n; j++)
    {
        if (arr[j] < arr[min])
        {
            min = j;
        }
    }
    temp = arr[i];
    arr[i] = arr[min];
    arr[min] = temp;
}

```

Prof. Vinayak Manjrekar

Our Centre

Thane - 9967504602

Nerul - 7506379305

Dadar - 9867674602

Vileparle - 8097435788

Borivali - 9821212420

Dombivli - 9322401430

y = 0x10
hexadecimaly = 010
octaly = 10
decimalASCII

A → 65

a → 97

zero → 0 → 48

space → 32

ASCII

85	Q	65	A	112	P	97	a
86	R	66	B	113	q	98	b
87	S	67	C	114	r	99	c
88	T	68	D	115	s	100	d
89	U	69	E	116	t	101	e
90	V	70	F	117	u	102	f
	W	71	G	118	v	103	g
	X	72	H	119	w	104	h
	Y	73	I	120	x	105	i
	Z	74	J	121	y	106	j
		75	K	122	z	107	k
		76	L			108	l
		77	M			109	m
		78	N			110	n
		79	O			111	o
		80	P				

Syllabus

No.	Module	Details Content	Hrs
1	Introduction to Computer, Algorithm And Flowchart	1.1 Basic Of Computer: Turning Model, Von Neumann Model, Basic Of positional Number System, Introduction to Operating System. 1.2 Algorithm And Flowchart: Three Construct Of Algorithm and Flowchart: Sequence, Decision(Selection) and Repetition	6
2	Fundamentals Of C-Programming	2.1 Character Set, Identifiers and Keywords, Data types, Constants, Variables. 2.2 Operators- arithmetic, Relational and logical, Assignment, Unary, Conditional, Bitwise, Comma, Other operators. Expression, Statements, Library Functions, Preprocessor. 2.3 Data Input Output- getchar (), putchar (), scanf (), printf (), gets (), puts (), Structure Of C Program.	6
3	Control Structures	3.1 Branching – If Statement, If else Statement , Multiway decision. 3.2 Looping – While, do- while, for 3.3 Nested control structure- Switch statement, Continue statement Break Statement, Goto statement.	12
4	Functions and Parameter	4.1 Function- Introduction of Function, Function Main, Defining a Function, Accessing a function, Function Prototype, passing Arguments to a Function, Recursion. 4.2 Storage Classes- Auto, Extern, Static, Register	6
5	Arrays, String Structure and Union	5.1 Array- Concepts, Declaration, Definition, Accessing array elements, One-dimensional and multidimensional array. 5.2 String:- Basic of string, array of String Functions in String.h 5.3 Structure- Declaration, Initialization, structure within structure, Operation on structures, Array of Structure. 5.4 Union- Definition, Difference between structure and union, operations on a union	14
6	Pointer and Files	6.1 Pointer :- Introduction, Definition and uses of Pointers, Address Operator , Pointer variables , Dereferencing Pointer, Void Pointer, Pointer Arithmetic, Pointers to Pointers, and Array, Passing Array to Function, Pointers and Function, Pointers and Two dimensional Array, Array of Pointers, Dynamic Memory Allocation. 6.2 Files:- Types of File, File operation- opening Closing , Creating, reading, Processing File.	8

STRUCTURED PROGRAMMING APPROACH		
INDEX		
1	Introduction to Computer, Algorithm And Flowchart	1 - 9
2	Basics of C	10 - 36
3	Expressing Algorithms - Selection	35 - 62
4	Expressing Algorithms - Iteration	43 - 56
5	Decomposition of Solution - Functions	55 - 68
6	Array	69 - 81
7	Strings	82 - 87
8	Structures and unions	88 - 95
9	File Management in C	96 - 100
10	Solution	Complete NB PDFs + files
	APPENDIX	—

Ch. 1. Introduction to Computer, Algorithm, Flowchart

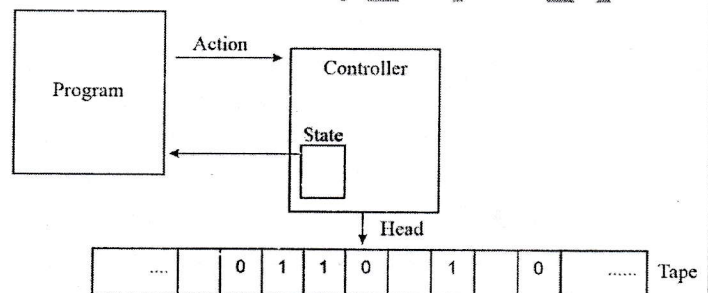
Turing Machine

Turing machine:- A Turing machine is a hypothetical machine thought of by the Mathematician Alan Turing in 1936. Despite its simplicity, the Machine can simulate ANY Computer algorithm, no matter how complicated it is!



A very simple representation of a Turing machine. It consists of infinitely-long tape which acts like the memory in a typical computer, or any other form of data storage. The squares on the tape are usually blank at the start and can be written with symbols. In this case, the machine can only process the symbols 0 and 1 " " (Blank), and is thus said to be a 3- symbols Turing machine. At any one time, the machine has a head which is positioned over one of the squares on the tape. With this head, the machine can perform three very basic operations:

- 1 read the symbol on the square under the head.
- 2 edit the symbol by writing a new symbols or erasing it.
- 3 move the tape left of right by one square so that the machine can read and edit the symbols on a neighbouring square.



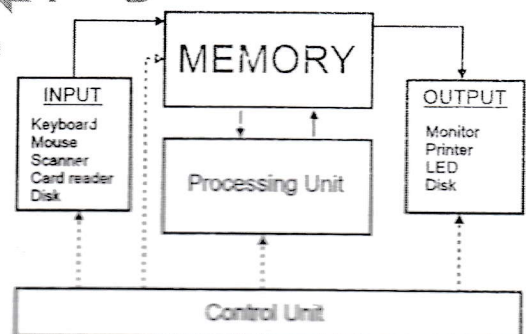
The Von Neumann Computer Model

- Von Neumann computer systems contain three main building blocks:
- the central processing unit (CPU),
- memory,
- input/output devices (I/O).

❖ These three components are connected together (using the system bus).

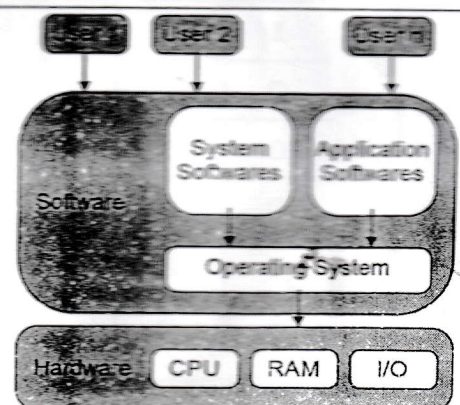
Components of the Von Neumann Model

1. Memory: Storage of information (data/program)
2. Processing Unit: Computation/Processing of Information
3. Input: Means of getting information into the computer. e.g. keyboard, mouse
4. Output: Means of getting information out of the computer. e.g. printer, monitor.
5. Control Unit: Makes sure that all the other parts perform their tasks correctly and at the correct time.



The Operating system (OS)

The Operating system (OS) is the most important program that runs on a computer. It is an interface between computer user and computer hardware. Every general-purpose computer must have an operating system to run other programs and applications. Computer operating Systems performs basic tasks, such as recognizing input from the keyboard, sending output to the display screen, keeping track of files and directories on the disk, and controlling peripheral device such as printers.



F.E. SEM – II STRUCTURED PROGRAMMING APPROCH

Operating systems provide a software platform on top of which other programs, called application programs, can run. The application programs must be written to run on top of a particular operating system. Your choice of operating system, therefore, determines to a great extent the application you can run. PCs, the most popular operating systems are DOS, OS/2, and windows But others are available such as Linux.

Interacting with the operating systems :-

As a users, you normally interacting with the operating systems though a set of commands.

For example, the DOS operating system contents commands such as COPY and RENAME For Copying Files and changing the names of files, respectively.

The commands are accepted and executed by a part of operating system called as command processor or command line interpreter. Graphical user interfaces allow you to commands by pointing and clicking at objects that appear on the screen. Popular operating systems The three most popular type of operating systems for personal and business computing include Linux, windows and MAC.

Linux Operating Systems

Linux is freely distributed open source operating systems that run on number of hardware platforms. The Linux kernel was develop mainly by Linus torvalds and it is based on Unix.

Windows Operating systems

Microsoft windows is a family of operating Systems for personal and business computers. Windows dominates the personal computers world, offering a graphical users interface (GUI), Virtual memory management, multitasking, and support for many peripheral devices.

MAC Operating systems

MAC OS is official name of the apple Macintosh operating systems. MAC OS features a graphical users Interface (GUI) that utilizes windows, icons and all applications that run on a Macintosh computers have a similar users interface.

Classification of Operating systems:-

- **Multi-user:-** Allows two or more users to run programs at the same time. Some operating systems permit hundreds or even thousands of concurrent users.
- **Multiprocessing:-** Supports running a program on more than one CPU.
- **Multitasking:-** Allows more than one programs to run Concurrently.
- **Multithreading:-** Allows Different parts of a single program to run concurrently.
- **Real time:-** Responds to input instantly. General -purpose operating systems, such as DOS and UNIX, Are not real time.

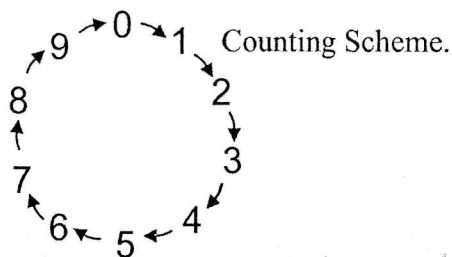
Positional Number System

1

1. Decimal Number System:-

Radix/ Base = 10

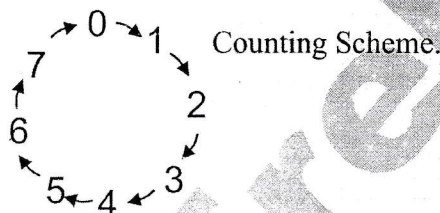
Symbol used = 0,1,2,3,4,5,6,7,8,9.



2. Octal Number system:-

Radix/ Base = 8

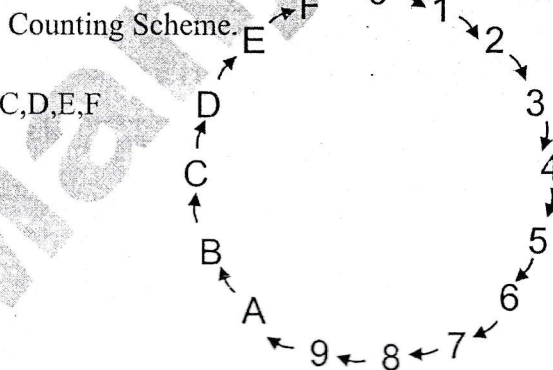
Symbol used = 0,1,2,3,4,5,6,7.



3. Hexadecimal Number system:-

Radix/ Base = 16

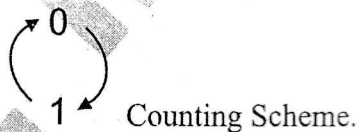
Symbol used = 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F



4. Binary Number system:-

Radix/ Base = 2

Symbol used = 0,1



Conversion (Deci-Bin)

$$25_{10} = 11001_2$$

2	25	
2	12	1
2	6	0
2	3	0
2	1	1
	0	1



$$98_{10} = 1100010_2$$

2	98	
2	49	0
2	24	1
2	12	0
2	6	0
2	3	0
2	1	1
	0	1



Positional Number System

2

Conversions:-

Decimal - Octal

8	125	
8	15	5
8	1	7
	0	1

$$125_{10} = 175_8$$

8	511	
8	63	7
8	7	7
	0	7

$$511_{10} = 777_8$$

Conversions:-

Decimal - Hexadecimal

16	254	
16	15	14
	0	15

$$254_{10} = FE_H$$

16	182	
16	11	6
	0	11

$$182_{10} = B6_H$$

Binary - Decimal

$$110101_2 = ?_{10}$$

$$(1 \times 2^5) + (1 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0)$$

$$32 + 16 + 0 + 4 + 0 + 1 = 53$$

$$110101_2 = 53_{10}$$

$$1000001_2 = ?_{10}$$

$$(1 \times 2^6) + (0 \times 2^5) + (0 \times 2^4) + (0 \times 2^3) + (0 \times 2^2) + (0 \times 2^1) + (1 \times 2^0)$$

$$64 + 1 = 65$$

$$1000001_2 = 65_{10}$$

Octal - Decimal

$$127_8 = ?_{10}$$

$$(1 \times 8^2) + (2 \times 8^1) + (7 \times 8^0)$$

$$64 + 16 + 7 = 87$$

$$127_8 = 87_{10}$$

$$362_8 = ?_{10}$$

$$(3 \times 8^2) + (6 \times 8^1) + (2 \times 8^0)$$

$$192 + 48 + 2 = 242$$

$$362_8 = 242_{10}$$

Hexadecimal - Decimal

$$2AC_H = ?_{10}$$

$$(2 \times 16^2) + (10 \times 16^1) + (12 \times 16^0)$$

$$512 + 160 + 12 = 684_{10}$$

$$2AC_H = 684_{10}$$

$$3E2_{16} = ?_{10}$$

$$(3 \times 16^2) + (14 \times 16^1) + (2 \times 16^0)$$

$$768 + 224 + 2 = 994_{10}$$

$$3E2_{16} = 994_{10}$$

Problem Definition: - In Computer programming i.e. when making a program we should have well defined problem. In computer programming, the term problem is the task to be performed. This problem has to be defined precisely so as to ensure the proper solution. This is one of the very important aspects of the problem solving, that the problem should be well defined. The problem defining has to undergo a various stages. The problem statement development begins from initially defining the problem statement. The problem once defined, and then the attempt should be made to solve the problem. You will come across many new things to be considered and implemented and hence you can redefine the problem statement accordingly. Thereafter scale the problem, for a larger data set and hence define the problem statement more precisely. Thus the problem statement can be defined and refined using the process. Once the problem is precisely defined then an algorithm is required to implement the same. The algorithm can then be implemented using a programming language.

Ex-1:- Write a Program to calculate permutation & Combination nP_r and nC_r . The value of nP_r and nC_r is given as, ${}^nP_r = \frac{n!}{(n-r)!}$ and ${}^nC_r = \frac{n!}{r!(n-r)!}$ Except the values of n & r from user.

Ex- 2 Accept a number from user and write a program to check whether the entered number is an Armstrong number. An Armstrong number is one, for which the sum of the cubes of each digit of the number is equal to the number itself. For instance, $153 = 1^3 + 5^3 + 3^3$

Ex- 3 Accept numbers from user & arrange them into an array. The number to be searched is entered through the keyboard by the user. Write a program to find if the number to be searched is present in the array and if it is present, display the number of times it appears in the array, also find out how many of the entered numbers are positive.

What is an Algorithm?

Program = Algorithms + Data

An algorithm is a part of the plan for the computer program. In fact, an algorithm is 'an effective procedure for solving a problem in a finite number of steps.'

Here is an example of an algorithm, for making a pot of tea.

1. If the kettle does not contain water, then fill the kettle.
2. Plug the kettle into the power point and switch it on.
3. If the teapot is not empty, then empty the teapot.
4. Place tea leaves in the teapot.
5. If the water in the kettle is not boiling, then go step 5.
6. Switch off the kettle.
7. Pour water from the kettle into the teapot.

It can be seen that the algorithm has a number of steps and that some steps (steps 1, 3, and 5) involve decision making and one step (step 5 in this case) involves repetition, in this case the process of waiting for the kettle to boil.

From this example, it is evident that algorithms show these three features:

- *Sequence (also known as process)*
- *Decision (also known as selection)*
- *Repetition (also known as iteration or looping)*

F.E. SEM - II STRUCTURED PROGRAMMING APPROCH

Therefore, an algorithm can be stated using three basic constructs: sequence, decision, and repetition.

Ex:2:- Print the largest number among three numbers

```
1. START
2. PRINT "ENTER THREE
   NUMBERS"
3. INPUT A, B, C
4. IF A >= B AND A >= C
   THEN PRINT A
5. IF B >= C AND B >= A
   THEN PRINT B
   ELSE
   PRINT C
6. STOP
```

Ex:3:- This algorithm uses a variable MAX to store the largest number.

```
1. START
2. PRINT "ENTER THREE NUMBERS"
3. INPUT A, B, C
4. MAX ← A
5. IF B > MAX
   THEN MAX ← B
6. IF C > MAX
   THEN MAX ← C
7. PRINT MAX
8. STOP
```

Ex:4:- Here, the algorithm uses a nested if construct.


```
1. START
2. PRINT "ENTER THREE
   NUMBERS"
3. INPUT A, B, C
4. IF A > B THEN
   IF A > C THEN
   PRINT A
   ELSE
   PRINT C
   ELSE IF B > C THEN
   PRINT B
   ELSE
   PRINT C
5. STOP
```

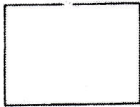
F.E. SEM – II STRUCTURED PROGRAMMING APPROACH

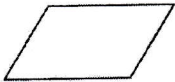
Flowcharts: -A flowchart provides appropriate steps to be followed in order to arrive at the solution to a problem. It is a program design tool which is used before writing the actual program. Flowcharts are generally developed in the early stages of formulating computer solutions.

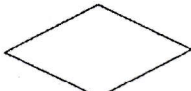
A flowchart comprises a set of various standard shaped boxes that are interconnected by flow lines. Flow lines have arrows to indicate the direction of the flow of control between the boxes. The activity to be performed is written within the boxes in English. **Standards for flowcharts** The following standards should be adhered to while drawing flow charts.


- Flowcharts start on the top of the page and flow down and to the right.
- Only standard flowcharting symbols should be used. English should be used in flowcharts, not programming language.
- The flowchart for each subroutine, if any, must appear on a separate page. Each subroutine begins with a terminal symbol with the subroutine name and a terminal symbol labelled return at the end.
- Draw arrows between symbols with a straight edge and use arrowheads to indicate the direction of the logic flow.


Start or end of the program or flowchart

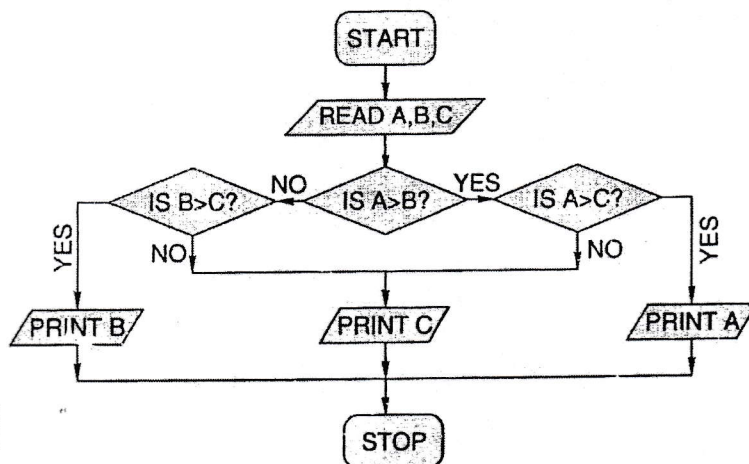

Computational steps or processing function of a program


Input entry or output display operation


A decision making and branching operation that has two alternatives

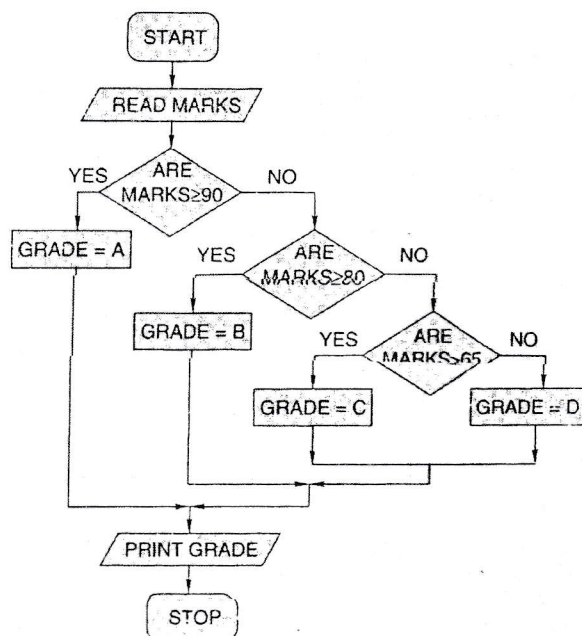

Connects remote parts of the flowchart on the same page

Draw the flowchart to find the largest of three numbers A, B, and C.

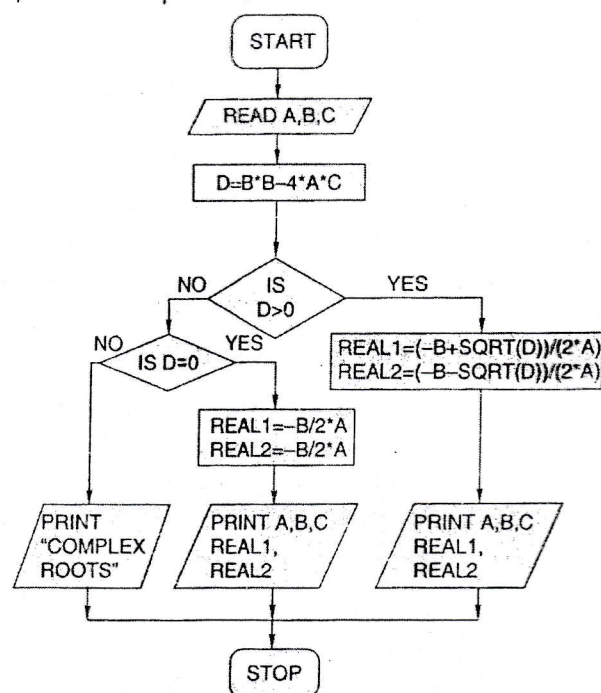


F.E. SEM - II STRUCTURED PROGRAMMING APPROACH

Prepare a flowchart to read the marks of a student and classify them into different grades. If the marks secured are greater than or equal to 90, the student is awarded Grade A; If they are greater than or equal to 80 but less than 90, Grade B is awarded; If they are greater than or equal to 65 but less than 80, Grade C is awarded; otherwise Grade D is awarded.

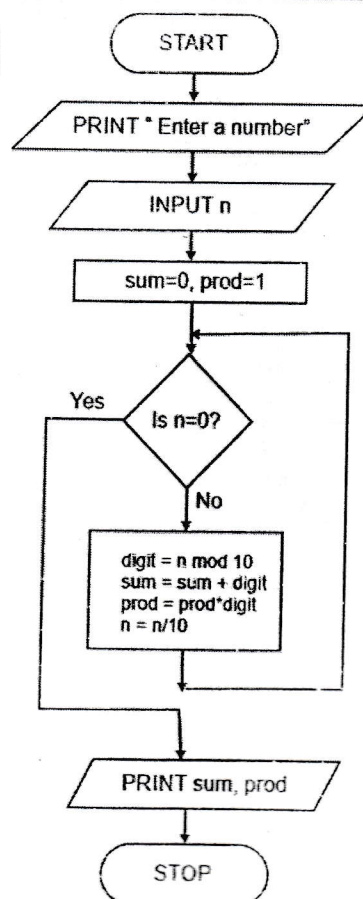


Draw a flowchart to find out the roots of quadratic equation.



Algorithm & flowchart to find the sum & product of all the digit of number using while loop.

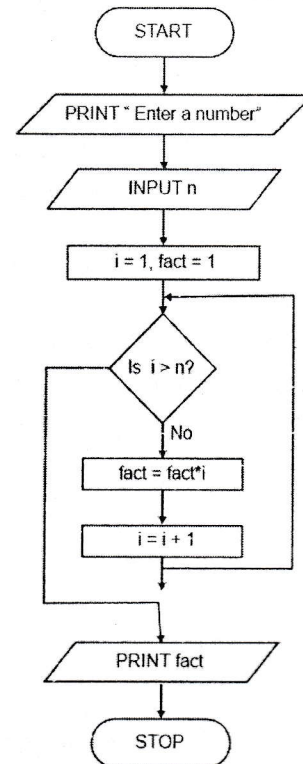
- Step 1: START
- Step 2: PRINT "Enter a number".
- Step 3: INPUT n.
- Step 4: sum = 0, prod = 1
- Step 5: IF n = 0, THEN GOTO step 10.
- Step 6: digit = n mod 10.
- Step 7: sum = sum + digit
- Prod = prod * digit
- Step 8: n = n / 10
- Step 9: GOTO step 5
- Step 10: PRONT sum , prod.
- Step 11: STOP.



F.E. SEM – II STRUCTURED PROGRAMMING APPROACH

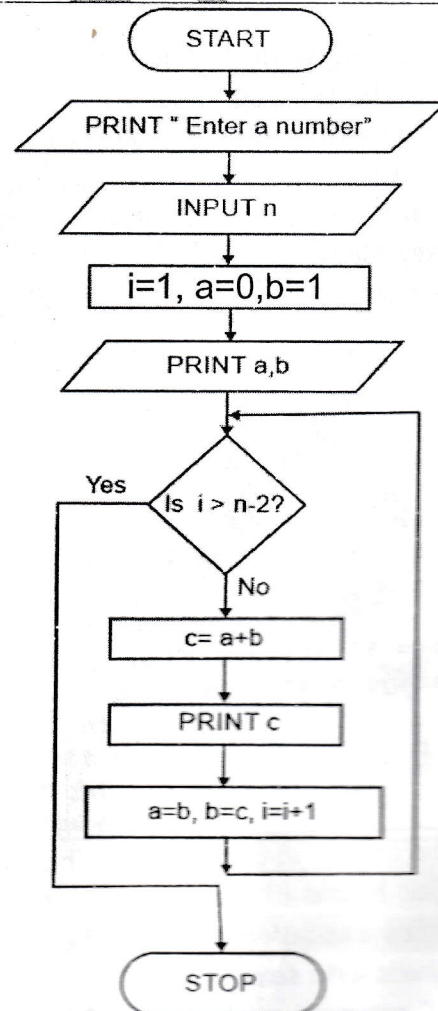
Algorithm & flowchart to find the factorial of a number.

Step 1: START
 Step 2: PRINT "Enter a number".
 Step 3: INPUT n.
 Step 4: $i = 1$, fact = 1
 Step 5: IF $i > n$ THEN GOTO step 9.
 Step 6: fact = fact * i.
 Step 7: $i = i + 1$
 Step 8: GOTO step 5
 Step 9: PRINT fact
 Step 10: STOP.



Algorithm & flowchart to display first n Fibonacci numbers.

Step 1: START
 Step 2: PRINT "Enter a number".
 Step 3: INPUT n.
 Step 4: $i=1$, $a=0$, $b=1$
 Step 5: PRINT a,b
 Step 6: IF $i > n-2$, THEN GOTO step 12
 Step 7: $c = a + b$
 Step 8: PRINT c
 Step 9: $a=b$, $b=c$.
 Step 10: $i=i+1$
 Step 11: GOTO step 6
 Step 12: STOP



CH 2. Basics of C

2.1 Importance of C:- 'C' is one of the most popular computer languages today because it is a structured, high-level, machine independent language. The increasing popularity of C is probably due to its many desirable qualities. It is most robust language whose rich set of built-in functions and operators can be used to write any complex program. The C compiler combines the capabilities of an assembly language with the features of a high-level language and therefore it is well suited for writing both system software and business packages.

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 and its strength lies in its built-in functions. Several standard functions are available which can be used for developing programs. 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 structured programming, thus requiring the use to think of a problem in terms of function modules or blocks. A proper collection of these modules would make a complete program. This modular structure makes program debugging, testing and maintenance easier.

Another important feature of C is its ability to extend itself. A C program is basically a collection of functions that are supported by the C library. We can continuously add our own functions to C library. With the availability of a large number of functions, the programming task becomes simple.

2.2 Getting started with C:- The classical method of learning English is to first learn the alphabets used in the language, then learn to combine these alphabets to form words, which in turn are combined to form sentences and sentences are combined to form paragraphs. Learning C is similar and easier. Instead of straight-away learning how to write programs, we must first know what alphabets, numbers and special symbols are used in C, then how using them, constants, variables and keywords are constructed, and finally how are these combined to form an instructions which would be combined later to form a program.

```
C2-P1:-
#include<stdio.h>
int main()
{
    float num1,num2,sum,ave;
    clrscr();
    printf("Enter 2 no: ");
    scanf("%f %f", &num1,&num2);
    sum=num1+num2;
    ave=sum/2;
    printf("sum = %f \n",sum);
    printf("average = %f",ave);
    getch();
    return(0);
}
```

Output:-

```
Enter 2 no: 5 10
sum = 15
average = 7.500000
```

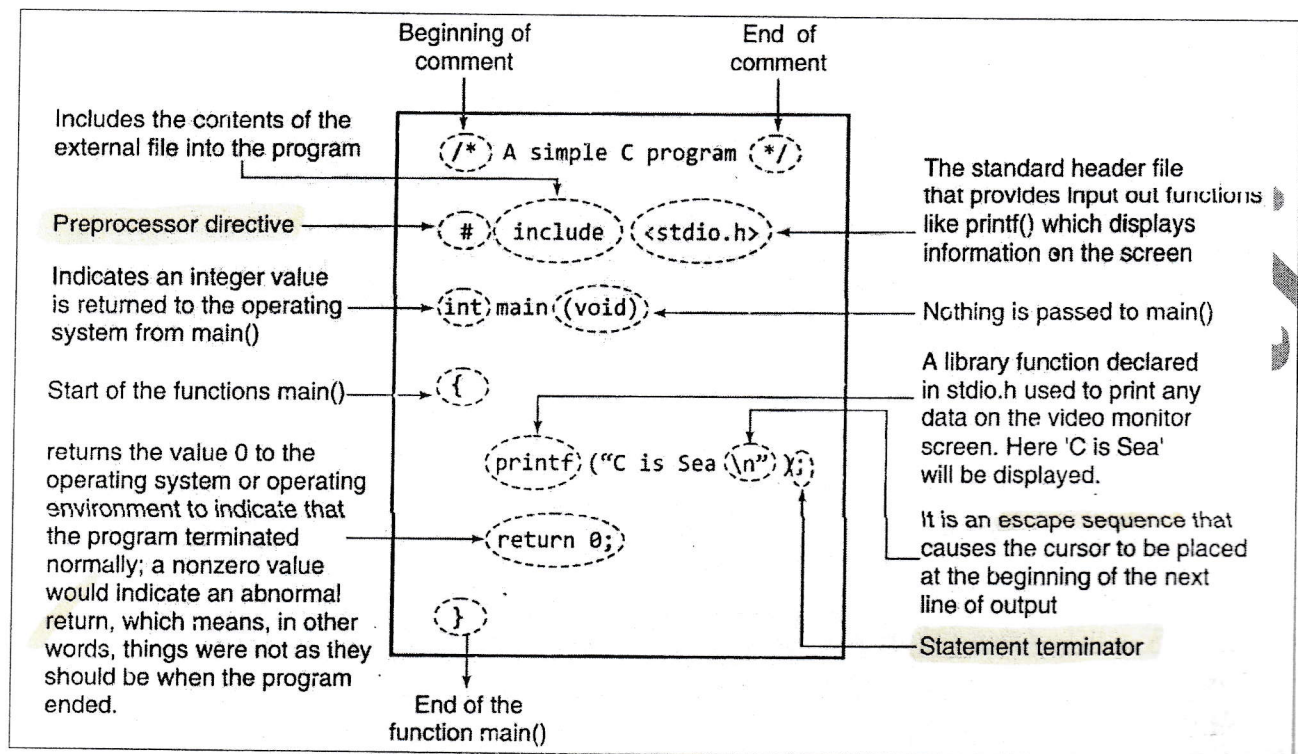
The C program is a collection of functions. The above example contains only one function, main(). As usual, execution begins at main(). Every C program must have a main(). C is a free from language. With a few exceptions, the compiler ignores carriage return and white spaces. The C statements terminate with semicolons.

Comments:- The C comment symbols /*----*/ are suitable for multiline comments. The following comment is allowed:

/* This is an example of

C program to illustrate Some of its features.*/

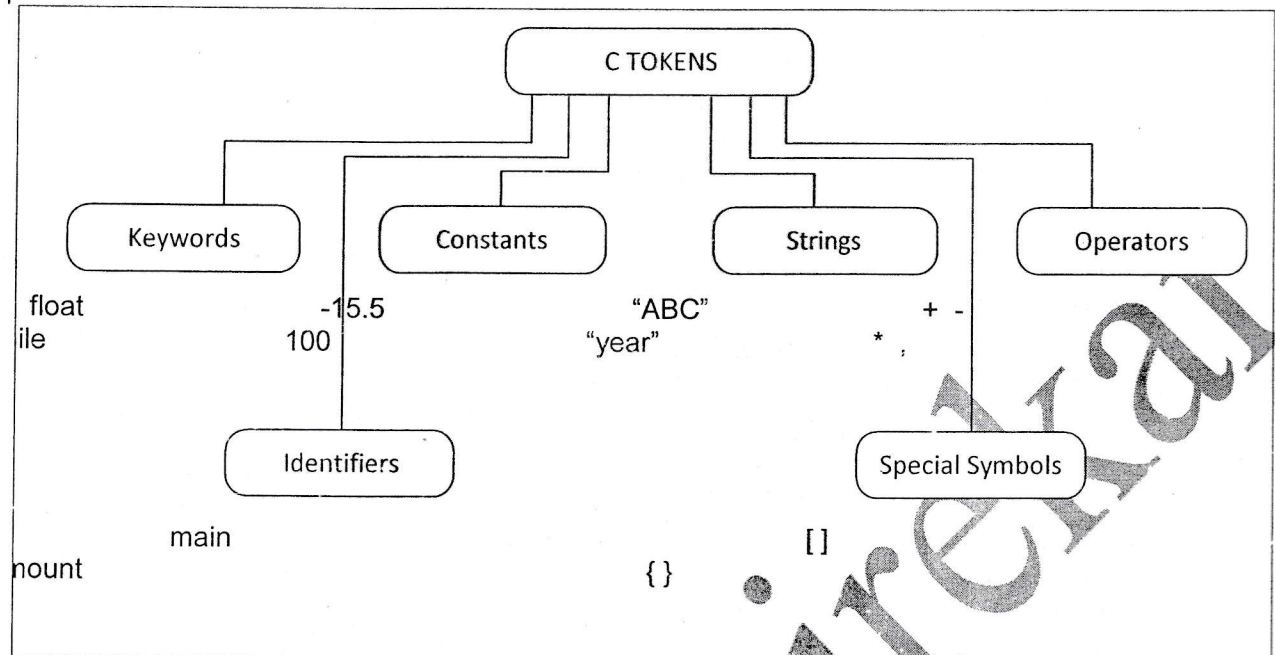
Return Type of main(): In above C program, main() returns an integer type value to the operating system. Therefore, main() should end with a return (0) statement; otherwise a warning or an error might occur. In some compiler main() can also be declared to return void.



2.3 C Character Set

Letters Uppercase A.....Z Lowercase a.....z	Digits All decimal digits 0.....9
Special Characters	
, comma . period ; semicolon : colon ? question mark \ apostrophe " quotation mark ! exclamation mark Vertical bar / slash \ backslash ~ tilde _ underscore \$ dollar sign % percent sign	& ampersand ^ caret * asterisk - minus sign + plus sign < opening angle bracket (or less than sign) > closing angle bracket (or greater than sign) (left parenthesis) right parenthesis [left bracket] right bracket { left brace } right brace # number sign
White Spaces Blank space Horizontal tab Carriage return New line Form feed	

2.4 C TOKENS:- In a passage of text, individual words and punctuation marks are called tokens. Similarly, in a C program the smallest individual units are known as C tokens. C has six types of tokens.

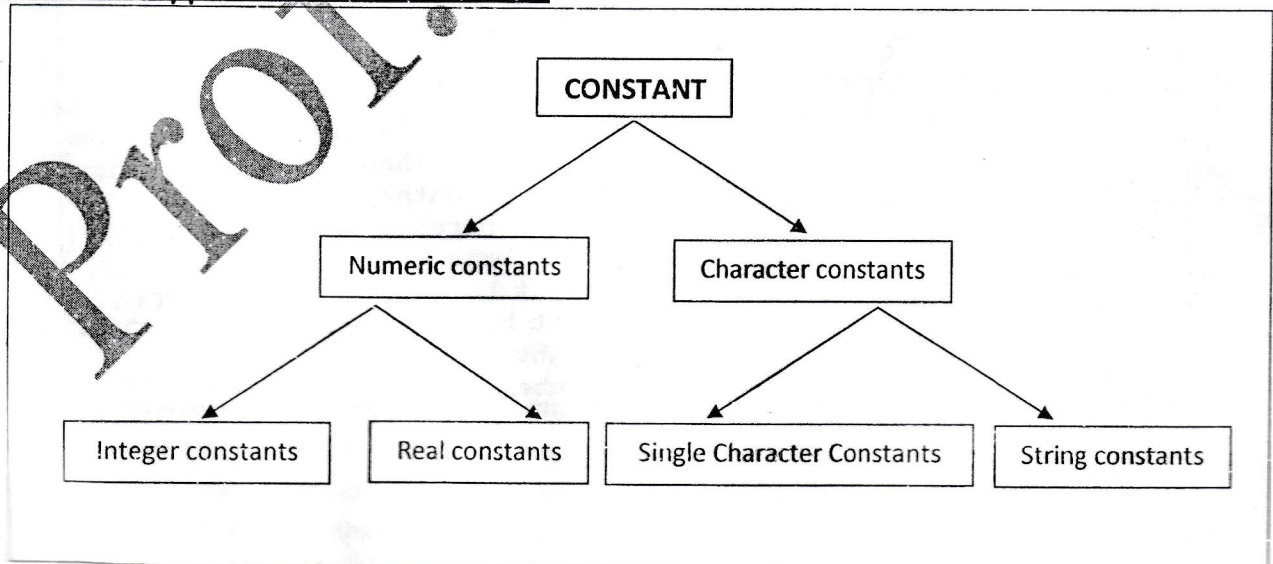


Keywords and Identifiers:- Every C word is classified as either a keyword or an identifier. All keywords have fixed meanings and these meanings cannot be changed. Keywords serve as basic building blocks for program statements.

Identifiers refer to the names of variables, functions and arrays. These are user-defined names and consist of a sequence of letters and digits, with a letter as a first character. Both uppercase and lowercase letters are permitted, although lowercase letters are commonly used. The underscore character is also permitted in identifiers. It is usually used as a link between two words in long identifiers.

Constants in C refer to fixed values that do not change during the execution of a program. C supports several types of constants.

2.5 Basic types of C constants:-



2.6 Backslash Character Constants:- C supports some special backslash character constants that are used in output functions. Note that each one of them represents one character,

although they consist of two characters. These characters combinations are known as escape sequences.

Constants	Meaning
'\a'	audible alert (bell)
'\b'	back space
'\f'	form feed
'\n'	new line
'\r'	carriage return
'\t'	horizontal tab
'\v'	vertical tab
'\"'	single quote
'\"'	double quote
'\?'	question mark
'\\'	backslash
'\0'	null

2.7 Rules for Constructing Integer constants:-

- 1) An integer constant must have at least one digit.
- 2) It must not have a decimal point.
- 3) It can be either positive or negative.
- 4) If no sign precedes integer constants it is assumed to be positive.
- 5) No commas or blanks are allowed within an integer constant.
- 6) The allowable range for integer constants is -32768 to 32767

Ex: 426 +782 - 8000 +7605

2.8 Rules for Constructing Real Constants:-

Real constants are often called Floating point constants. The real constants could be written in two forms - Fractional form and Exponential form.

Following rules must be observed while constructing real constants expressed in fractional form:-

- a) A real constant must have at least one digit.
- b) It must have a decimal point.
- c) It could be either positive or negative.
- d) Default sign is positive.
- e) No commas or blanks are allowed within a real constant.

Ex: +325.34, 426.0, -32.76, 48.5792,

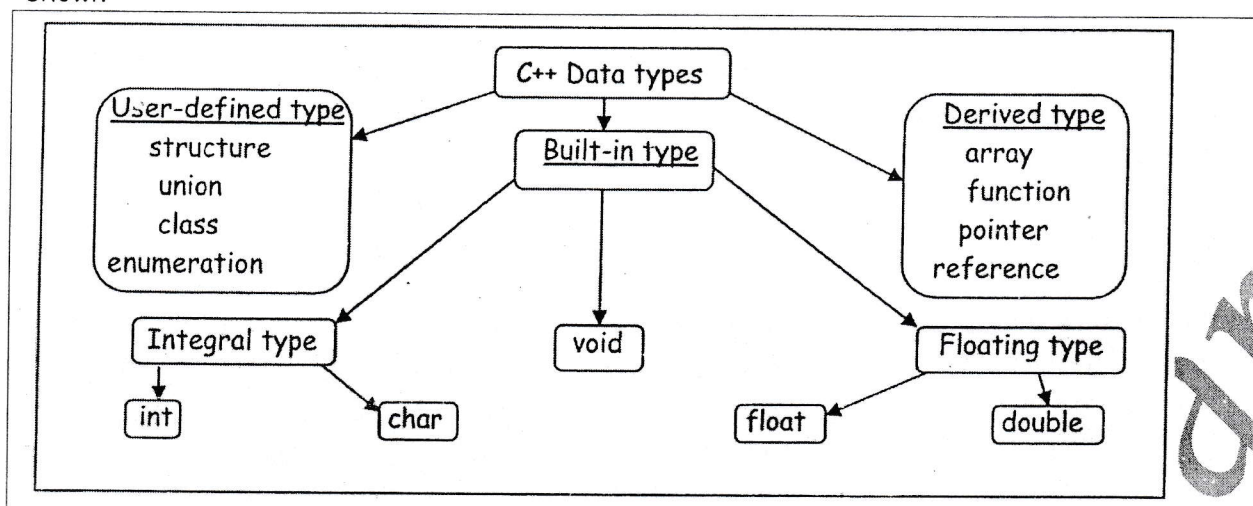
2.9 Rules for constructing character constants:-

- a) A character constant is a single alphabet, a single digit or a single special symbol enclosed within a single inverted commas.
- b) The maximum length of a character constant can be 1 character. Ex- 'A' 'T' '5' '\'

2.10 C Keywords:- Keywords are the words whose meaning has already been explained to the C compiler. The keywords cannot be used as variables names. The keywords are also called 'Reserved words'.

Keywords	Description
① auto	Declares an automatic variable, its memory is allocated by the compiler for the duration of its existence.
break	Terminates the execution of the <u>closest</u> enclosing loop or conditional statement in which it appears.
case	Allow execution of list of groups of statements.
char	Represents a character data type.
const	Specifies that the object or a variable have a constant value that cannot be changed.
default	Used with switch statement that indicates the specified action should be taken, if no other actions are appropriate.
do	Used with while statement to <u>construct a loop that needs to be executed at least once.</u>
double	Represents a real-number value.
else	Used with if, to execute set of statements according to the results of a comparison statement.
② extern	Indicates that a statement used in one program module is <u>defined in different program module.</u>
float	Indicates a floating-point, real number data element.
for	Constructs a loop that needs to execute for a specified number of times.
goto	Allows switching to a specified place in a function.
if	Executes statement if a condition is true.
→ long	Indicates a long version of a particular data type to be used.
main	Indicates the starting position of execution for all the C and C.
③ register	Specifies that the variable is to be stored in a CPU register and not in the memory.
return	Terminates the execution of a function and return control to the calling function.
→ short	Specifies a shorter version of a particular data type with a smaller range of values to be used.
→ signed	Specifies that the data type accept positive and negative values.
sizeof	Specifies the amount of the storage in bytes, required to store an object of the type of the operand.
④ static	Specifies that the variable has static duration and initialized it to zero, unless some other value is initialized.
struct	Specifies a user-defined type that comprises of members of different data types.
switch	Allows selection among multiple selection of code depending on the value of the expression.
typedef	Declares a user - define data type.
union	Defines a user-defined data or class type that contains only one object from its list of members.
→ unsigned	Indicates that the variable only takes positive values.
void	Specifies that the function does not return a value.
volatile	Indicates that a variable or an object can be modified by another part of the program.
while	Executes statements repeatedly, until expression evaluates to zero.

2.11 BASIC DATA TYPES:- Data types in C can be classified under various categories as shown



2.12 Size and range of C Basic Data type for 16 bit computer

Type	Bytes	Range
Char or signed char	1	-128 to 127
unsigned char	1	0 to 255
int or signed int	2	-32768 to 32767
unsigned int	2	0 to 65535
short int	2	-32768 to 32767
unsigned short int	2	0 to 65535
long int	4	-2147483648 to 2147483647
signed long int	4	-2147483648 to 2147483647
unsigned long int	4	0 to 4294967295
Float	4	-3.4e38 to 3.4e38
Double	8	-1.7e308 to 1.7e308
long double	10	-3.4e4932 to 3.4e4932

DATA INPUT AND OUTPUT

printf() and scanf(): The functions printf() and scanf() fall under the category of formatted console I/O functions. These functions allow us to supply the input in a fixed format and let us obtain the output in the specific form.

2.13 printf()

The general form of printf statement is,

```
printf("<format string>", <list of variables>);
```

The format specifier can obtain:

1. Character that are simply printed as they are.
2. Conversion specification that begin with a % sign.
3. Escape sequences that begin with a \ sign.

DATA TYPE	FORMAT SPECIFIER / CONVERSION CHARACTER
Character	%c
String	%s
Integer - decimal	%d

Integer - octal	%o
Integer - hexadecimal	%x
Long integer - decimal	%ld
Long integer - octal	%lo
Long integer - hexadecimal	%lx
Float	%f
Double	%lf
Long double	%Ld

Printf() with Field width specifier:- The field width specifier tells printf() how many columns on screen should be used while printing a value. For example, %10d says "print the variable as a decimal integer in a field of 10 columns." If the value to be printed happens not to fill up the entire field, the value is right justified and is padded with blanks on the left. If we include the minus sign in conversion specification (as in %-10d), this means left justification is desired and the value will be padded with blanks on the right.

Some systems also support a special field specification character that lets the user define the field size at the run time. This takes the following form:

```
printf("%*.*", width, precision, number);
```

In this case, both the field width and the precision are given as arguments which will supply the values for w and p. For example,

```
printf("%*.*f", 7, 2, number);
```

is equivalent to

```
printf("%7.2f", number);
```

The advantage of this format is that the values for width & precision may be supplied at run time thus making the format a dynamic one. For example, the above statement can be used as follows:

```
int width = 7;
int precision = 2;
.....
.....
printf ("%*.*f", width, precision, number);
```

2.14 Formatted Input – Scanf

Scanf() allows us to enter data from keypad that will be formatted in a certain way.

The general form of scanf() statement is as follows:

```
scanf("format string", list of addresses of variables);
```

For example:

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

Note that we are sending addresses of variables (addresses are obtained by using '&' the addresses of operator) to scanf function. This is necessary because the values received from keyboard must be dropped into variables corresponding to these addresses. The values supplied through keyboard must be separated by blanks(s), tab(s), or newline(s).

Consider the following example:

```
scanf ("%2d %5d", &num1, &num2);
```

suppose the Data is as follows:

50 31426

The variable num1 will be assigned 31 (because of %2d) and num2 will be assigned 426 (unread part of 31426). The value 50 that is unread will be assigned to the first variable in the next scanf call.

What happens if we enter a floating point number instead of an integer? The fractional part may be stripped away!

Also, scanf may skip reading further input. An input field may be skipped by specifying * in the place of field width. For example, the statement


```
scanf ("%d %d %d", &a, &b)
will assign the data
123 456 789
as follows:
123 to a
456 skipped (because of *)
789 to b
```

C2-P2:-

```
#include<stdio.h>
int main()
{ int x = 123;
clrscr();
printf("%d\n",x);
printf("%6d\n",x);
printf("%4d\n",x);
printf("%06d\n",x);
printf("%04d\n",x);
printf("%2d\n",x);
getch();
return(0); }
```

Output:

1	2	3			
			1	2	3
	1	2	3		
0	0	0	1	2	3
0	1	2	3		
1	2	3			

C2-P3:-

```
#include<stdio.h>
int main()
{ float x = 1.23456;
clrscr();
printf("%f\n",x);
printf("%12f\n",x);
printf("%.3f\n",x);
printf("%12.3f\n",x);
printf("%012.3f\n",x);
getch();
return(0);
}
```

Output:

1	.	2	3	4	5	6	0				
				1		2	3	4	5	6	0
1	.	2	3	5							
						1	.	2	3	5	
0	0	0	0	0	0	0	1	.	2	3	5

* C2-P4:- WAP to print ASCII value of any character. Nbk.

2.15 getch(), getche(), getchar(), putchar(), and putchar() functions :

We often want a function that will read a single character the instant it is typed without waiting for the enter key to be hit. getch() and getche() are the two functions which serve this purpose. These functions return the character that has been most recently typed. The 'e' in getche() function means it displays the character that you typed to the screen. As against this getch() just returns the character that you typed on the screen. getchar() work similarly and displays the character that you typed on screen, but unfortunately requires the enter key to be hit following the character to be typed.

Putch() and putchar() from the other side of the coin. They print a character on the screen. As far as the working is concern they are exactly same. The limitation of putchar() and putch() is that they can output only one character at a time.

* C2-P5:-

```
#include<stdio.h>
int main( )
{char ch='A';
clrscr();
putch(ch);
putchar(ch);
printf("\n Press any key to continue : ");
```



```

ch = getch( ); /*WILL NOT ECHO THE CHARACTER*/
printf("\n you entered : ");
putch(ch); /*WILL DISPLAY THE CHARACTER*/
printf("\n Type any character : ");
getche( ); /*WILL ECHO THE CHARACTER*/
printf("\n Type any character then press Enter: ");
getchar( ); /*echo the character but must be followed by enter key*/
getch( ); return(0);
}

```

Output:-

AA

Press any key to continue :
 you entered : j
 Type any character : p
 Type any character then press Enter: d

2.16 C Instructions:- There are basically three type of instruction in C:

- 1) Type Declaration Instruction.
- 2) Arithmetic Instruction.
- 3) Control Instruction.

Type Declaration Instruction:

The type declaration statement is written at the beginning of main() function.

[a] While declaring the type of variable we can also initialize it as shown below:

```

int i = 10, j = 25;
float a = 1.5, b = 1.99 + 2.4*1.44;

```

[b] The order in which we define the variables is sometimes important.

```
float b = a + 3.1, a = 1.5;
```

This statement is wrong.

[c] The following statements would work

```

int a,b,c;
a = b = c = 10;

```

However, the following statement would not work

```
int a = b = c = 10;
```

Once again we are trying to use b (to assign to a) before defining it.

Arithmetic Instruction: A C arithmetic instruction consists of a variable name on the left hand side of = and variable names & constants on the right hand side of =. The variables and constant appearing on the right hand side of = are connected by arithmetic operations like +, -, *, and /.

Ex:-

```

int ad;
float delta, alpha, beta, gamma;
ad = 3200;
delta = alpha* beta / gamma + 3.2* 215;

```

A C arithmetic statement could be three types. These are as follow:-

[a] Integer mode arithmetic statement This is an arithmetic statement in which all operands are either integer variable or integer constants.

Ex:-

```

int i, j, k;
i = i + 1;
j = k*234 + i - 7689;

```

[b] Real mode arithmetic statement- This is an arithmetic statement in which all operands are either real constant or real variables.

Ex- float

```

a,b,c,d,e,f;
a = b + 23.123 / 4.5*0.3442;
c = d*e*f / 100.0;

```


[c] Mixed mode arithmetic statement - This is an arithmetic statement in which some of the operands are integers and some of the operands are real.

```
Ex:- float p,q,r,s,avg;
int a, b, c, num;
p = q* r * s/ 100.0;
avg = (a+ b+ c + num)/4;
```

It is very important to understand how the execution of an arithmetic statement takes place. Firstly, the right hand side is evaluated using constants and the numerical values stored in the variable names. This value is then assigned to the variable on the left-hand side.

[.] C allows only one variable on left-hand side of =. That is, $z = k*1$ is legal, whereas $k*1 = z$ is illegal.

[.] In addition to the division operator, ^{which returns the quotient, the modulus operator} returns the remainder on dividing one integer with another. Thus the expression $10/2$ yields 5, whereas, $10\% 2$ yield 0. Note that the modulus operator (%) cannot be applied on a float. Also note that on using % the sign of the remainder is always same as the sign of the numerator. Thus $-5\% 2$ yields -1, whereas, $5\% -2$ yields 1.

[.] An arithmetic instruction is often used for storing character constants in character variables.

```
char a, b, d;
a = 'F';
b = 'G';
d = '+';
```

When we do this the ASCII values of the characters are stored in the variables. ASCII values are used to represent any character in memory. The ASCII values of 'F' and 'G' are 70 and 71.

[.] Arithmetic operations can be performed on ints, floats and chars. Thus the statements,

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

are perfectly valid, since the addition is performed on the ASCII values of the characters and not on character themselves. The ASCII values of 'a' and 'b' are 97 and 98, and hence can definitely be added.

2.17 L values and R values

An lvalue is an expression to which a value can be assigned. An rvalue can be defined as an expression that can be assigned to an lvalue. The lvalue expression is located on the side of an assignment, where as an rvalue is located on the right side of an assignment statement.

An lvalue cannot be a constant, for example, consider the following statements:-

```
1 = x;
x + y = a + b;
x + b = 5;
```

In each of the above cases, the left side of the statement do not represent storable locations in memory. Therefore, these two assignment statement do not contain lvalue and will generate compiler errors.

2.18 INCREMENT AND DECREMENT OPERATORS

C allows two very useful operators not generally found in other languages. These are the increment and decrement operators:

++ and --

The operator ++ adds 1 to the operand, while -- subtracts 1. Both are unary operators and takes the following forms:

++m; or m++;

--m; or m--;

++m; is equivalent to $m = m + 1$; (or $m += 1$;))

--m; is equivalent to $m = m - 1$; (or $m -= 1$;))

We use the increment and decrement statements in **for** and **while** loops extensively.

While ++m and m++ mean the same thing when they form statements independently, they behave differently when they are used in expressions on the right-hand side of an assignment statement.

Consider the following:

m = 5;

y = ++m;

In this case, the value of y and m would be 6. Suppose, if we rewrite the above statements as

m = 5;

y = m++;

then, the value of y would be 5 and m would be 6. A *prefix operator* first adds 1 to the operand and then the result is assigned to the variable on left. On the other hand, a *postfix operator* first assigns the value to the variable on left and then increments the operand.

Rules for ++ and -- Operators

- Increment and decrement operators are unary operators and they require variable as their operands.
- When postfix ++ (or --) is used with a variable in an expression, the expression is evaluated first using the original value of the value and the variable is incremented (or decremented) by one.
- When prefix ++(or --) is used in an expression, the variable is incremented (or decremented) first and then the expression is evaluated using the new value of the variable.
- The precedence and associativity of ++ and -- operators are the same as those of unary + and unary -.

SPECIAL OPERATORS

C supports some special operators of interest such as comma operator, size of operator, pointer operators (& and *) and member selection operators (. and ->).

2.19 The Comma Operator

The comma operator can be used to link the related expressions together. A comma-linked list of expressions are evaluated *left to right* and the value of *right-most* expression is the value of the combined expression. For example, the statement

value = (x = 10, y = 5, x+y);

first assigns the value 10 to x, then assigns 5 to y, and finally assigns 15 (i.e. 10+5) to value. Since comma operator has the lowest precedence of all operators, the parentheses are necessary. Applications of comma operator are:

In for loops: `for (n = 1, m = 10, n<=m; n++, m++)`

In while loops: `while (c = getchar(), c != '10')`

2.20 The size of Operator

The `sizeof` is a compile time operator and, when used with an operand, it returns the number of bytes the operand occupies. The operand may be a variable, a constant or a data type qualifier.

Examples:

`m = sizeof(sum);`

`n = sizeof(long int);`

The `sizeof` operator is normally used to determine the lengths of arrays and structures when their sizes are not known to the programmer.

2.21 Associativity of Operators:- When the expression contains two operators of equal priority the tie between them is settled using the associativity of the operators. Associativity can be of two types left to right or right to left. Left to right associativity means that the left operand must be unambiguous. It must not be involved in evaluation of any other sub-expression.

Similarly, in case of right to left associativity the right operand must be unambiguous.

Consider the expression,

`a = 3 / 2 * 5;`

Since both `/` and `*` have L & R associativity and only `/` has unambiguous left operand, it is performed earlier.

Consider one more expression,

`a = b = 3;`

Since both `=` have R to L associativity and only the second `=` has unambiguous right operand (necessary condition for R to L associativity) the second `=` is performed earlier.

2.22 Order of precedence of Operator in C:-

Type of Operator	Order
Unary	<code>++ -- ~ ! ++ --</code>
Multiplicative	<code>* / %</code>
Additive	<code>+ -</code>
Shift	<code><< >> >>></code>
Relational	<code><< >> < > <= >=</code>
Equality	<code>= !=</code>
Bit Wise	<code>& ^ </code>
Logical	<code>&& </code>
Conditional (ternary)	<code>? :</code>
Assignment	<code>= += -= *= /= % =</code>

2.23 Hierarchy of operation:-

Priority	Operators	Description	Ass.
1 st	<code>* / %</code>	Multiplication, division, modular division	L to R
2 nd	<code>+ -</code>	Addition, Subtraction	L to R
3 rd	<code>=</code>	Assignment	R to L

Stepwise valuation of $i = 2 * 3 / 4 + 4 / 4 + 8 - 2 + 5 / 8$ this expression is shown below:

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

$$i = 6 / 4 + 4 / 4 + 8 - 2 + 5 / 8$$

$$i = 1 + 4 / 4 + 8 - 2 + 5 / 8$$

$$i = 1 + 1 + 8 - 2 + 5 / 8$$

$$i = 1 + 1 + 8 - 2 + 0$$

$$i = 2 + 8 - 2 + 0$$

$$i = 10 - 2 + 0$$

$$i = 8 + 0$$

$$i = 8$$

Operation = *

Operation = /

Operation = /

Operation = /

Operation = +

Operation = +

Operation = -

Operation = +

Algebraic Expression	C Expression
$a \times b - c \times d$	$a * b - c * d$
$(m + n) (a + b)$	$(m + n) * (a + b)$
$3x^2 + 2x + 5$	$3 * x * x + 2 * x + 5$
$\frac{a + b + c}{d + e}$	$(a + b + c) / (d + e)$
$\left[\frac{2By}{d+1} - \frac{x}{3(z+y)} \right]$	$2 * b * y / (d + 1) - x / 3 * (z + y)$

2.24 Bitwise Operators

Operators	Meaning
&	bitwise AND
!	bitwise OR
^	bitwise exclusive OR (XOR)
~	one's complement (NOT)
<<	shift left
>>	shift right
>>>	shift right with zero fill

Functioning of Bit-wise Operators :- The following table shows the operation of the four Bitwise operators &, |, ^ and ~.

A	B	A B	A&B	A^B	~A
0	0	0	0	0	1
0	1	1	0	1	1
1	0	1	0	1	0
1	1	1	1	0	0

Bit-wise NOT -

The bit-wise operator (~) inverts all the bits of its operand. For example, the number 24 which has the following bit pattern, 00011000

becomes, 11100111

after the NOT (~) operator is applied.

The bit-wise NOT operator is also called as bit-wise complement operator.

The bitwise assignment operators can be listed as, &= |= ^=

Consider the following examples.

Operation	Equivalent Operation
$a \&= b;$	$a = a \& b;$
$c \ = \ d$	$c = c \ \ d;$
$e \ ^= \ f$	$e = e \ ^ \ f;$

Variables a,b,c,d,e& f are of type Boolean.

2.25 Bitwise Shift Operators:- The shift operators are used to move bit patterns either to the left or to the right. The shift operators are represented by the symbols \ll and \gg .

\ll **Left shift:-** The left-shift operation cause all the bits in the operand op to be shifted to the left by n positions. The leftmost n bits in the original bit pattern will be lost and rightmost n bits positions that are vacated will be filled with 0s.

\gg **Right shift:-** The right-shift operation cause all the bits in the operand op to be shifted to the right by n positions. The rightmost n bits will be lost. The leftmost n bit positions that are vacated will be filled with zero, if the op is positive integer. If the variable to be shifted is negative, then the operation preserves the high-order bit of 1 and shifts only the lower 31 bits to the right.

Let int $n =$ 0100 1001 1100 1011
Then, $n \ll 3 =$ 0100 1110 0101 1000
 $n \gg 3 =$ 0000 1001 0011 1001

↑ unsigned
& signed
concept

Shift operators are often used for multiplication and division by powers of two.

Consider the following statement:

$n = m \ll 1;$

This statement shifts one bit to the left in m and then the result is assigned to n . The decimal value of n will be the value of m multiplied by 2. Similarly, the statement

$n = m \gg 1;$

This statement shifts m one bit to the right and assigns the result to n . In this case, the value of n will be the value of m divided by 2.

$n = m \ll 2 \Rightarrow \text{mult by } 2^2$ $n = m \ll 4 \Rightarrow \text{mult by } 2^4$
 $n = m \gg 6 \Rightarrow \text{div by } 2^6$ $R.D. \text{ Rahul David}$

```
C2-P6:-
#include<stdio.h>
int main()
{ int a=6,b= 72;
clrscr();
printf("a = %d & b = %d \n",a,b);
printf("Right shift a by 1 :a>>1 = %d \n", (a>>1) );
printf("Left shift a by 1 :a<<1 = %d \n", (a<<1) );
printf("Right shift b by 3 :b>>3 = %d \n", (b>>3) );
printf("Left shift b by 3 :b<<3 = %d \n", (b<<3) );
getch();
return (0);
}
```

O/p: $a=6$ & $b=72$
Right shift a by 1: $a \gg 1 = 3$
Left shift a by 1: $a \ll 1 = 12$
Right shift b by 3: $b \gg 3 = 9$
Left shift b by 3: $b \ll 3 = 576$
 $a=6$ & $b=72$

⊗ values of a & b were not modified ⊗

2.26 Integer and Float Conversions:-

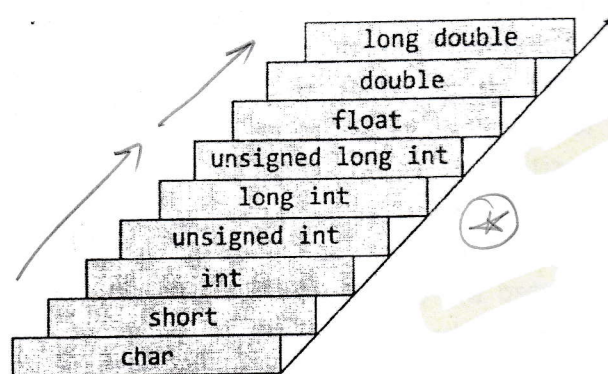
- [a] An arithmetic operation between an integer and integer always yield an integer result.
- [b] An operation between an integer and real always yields a real result.
- [c] An operation between an integer and real always yields a real result. In this operation the integer is first promoted to a real and then the operation is performed. Hence the result is real.

Operation	Result	Operation	Result
5/2	2	2/5	0
5.0/2	2.5	2.0/5	0.4
5/2.0	2.5	2/5.0	0.4
5.0/2.0	2.5	2.0/5.0	0.4

2.27 TYPE CONVERSIONS IN EXPRESSIONS

Implicit Type Conversion

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



During the evaluation it adheres to very strict rules of type conversion. If the operands are of different types, the 'lower' type is automatically converted to the 'higher' type before the operation proceeds. The result is of the higher type. eg) $5/2.0 = 5.0/2.0 = 2.5$

The final result of an expression is converted to the type of the variable on the left of the assignment sign before assigning the value to it. However, the following changes are introduced during the final assignment.

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

Arithmetic Instruction (K is integer)	Result	Arithmetic Instruction (a is float)	Result
$K = 2/9$	0	$a = 2/9$	0.0
$K = 2.0/9$	0	$a = 2.0/9$	0.2222
$K = 2/9.0$	0	$a = 2/9.0$	0.2222
$K = 2.0/9.0$	0	$a = 2.0/9.0$	0.2222
$K = 9/2$	4	$a = 9/2$	4.0
$K = 9.0/2$	4	$a = 9.0/2$	4.5
$K = 9/2.0$	4	$a = 9/2.0$	4.5
$K = 9.0/2.0$	4	$a = 9.0/2.0$	4.5

Explicit Conversion

We have just discussed how C performs type conversion automatically. However, there are instances when we want to force a type conversion in a way that is different from the automatic conversion.

int a=5, b=2;

float c = a/b;

① $c = (\text{float}) a/b;$ (5.0/2) ② $c = \text{float}(a/b);$ (5/2 to float)
o/p: $c = 2.500000$ o/p: $c = 2.000000$

Since a and b are declared as integers in the program, the decimal part of the result of the division would be lost and c would represent a wrong figure. This problem can be solved by converting locally one of the variables of the floating point as shown below:

$c = (\text{float}) a/b;$

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

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

(type-name)expression;

When *type-name* is one of the standard C data types. The expressions may be a constant, variable or an expression.

Use of Casts

Example	Action
<code>x = (int) 7.5</code>	7.5 is converted to integer by truncation.
<code>a = (int) 21.3/(int)4.5</code>	Evaluated as 21/4 and the result would be 5.
<code>b = (double) sun/n</code>	Division is done in floating point mode.
* <code>y = (int) (a+b)</code>	The result of a+b is converted to integer.
* <code>z = (int)a+b</code>	a is converted to integer and then added to b.
<code>p = cos((double)x)</code>	Converts x to double before using it

2.28 Defining Symbolic Constants:- We often use certain unique constants in a program. These constants may appear repeatedly in a number of places in the program. One example of such a constant is 3.142, representing the value of the mathematical constant "pi".

We may like to change the value of "pi" from 3.142 to 3.14159 to improve the accuracy of calculations. We will have to search throughout the program and explicitly change the value of the constant wherever it has been used. If any value is left unchanged, the program may produce disastrous outputs.

Assignment of such constants to a symbolic name frees us from these problems.

Valid examples of constant definitions are:-

```
#define STRENGTH 100
#define PASS_MARK 50
#define MAX 200
#define PI 3.14159
```

The following rules apply to a `#define` statement which define a symbolic constant.

- [1] Symbolic names have the same form as variable names. (Symbolic names are written in CAPITALS to visually distinguish them from the normal variable names, which are written in lowercase letters. This is only a convention, not a rule)
- [2] No blank space between the pound sign '#' and the word define is permitted.
- [3] '#' must be the first character in the line.
- [4] A blank space is required between `#define` and symbolic name and between the symbolic name and the constant.
- [5] `#define` statements must not end with a semicolon.
- [6] After definition, the symbolic name should not be assigned any other value within program by using an assignment statement. For example, `STRENGTH = 200;` is illegal.
- [7] Symbolic names are NOT DECLARED FOR DATA TYPES. Its data type depends on the type of constant.
- [8] `#define` statements may appear anywhere in the program but before it is referenced in the program (the usual practice is to place them in the beginning of the program).

Examples of Invalid # define statements.

Statement	Validity	Remark
<code>#define x = 2.5</code>	Invalid	'=' sign is not allowed
<code># define MAX_ARRAY10</code>	Invalid	No white space between # and define
<code>#define n 25;</code>	Invalid	No semicolon at the end
<code>#define N 5, M 10</code>	Invalid	A statement can define only one name

#Define ARRAY 11	Invalid	define should be in lowercase letters
#define PRICES\$ 100	Invalid	\$ symbol is not permitted in name

2.29 PREPROCESSOR DIRECTIVES:-

The pre-processor statements starts with a special character '#' and does not end with ';'.
 The pre-processor sees the program file before it is seen by the compiler. The pre-processor reads the statements in the program and identifies ^{ones} ~~one~~ that starts with the special character '#'. These statements are called as 'directives' because they gives direction to the pre-processor to take certain action after reading these directives.

The pre-processor thus after reading these directives, remove these from the source program and instead takes the actions accordingly as directed by the directives.

The pre-processor removes all the directives i.e. the statements that starts with '#', after taking actions accordingly, from the input source code. This source code without directives is passed onto the 'C' compiler.

The 'C' compiler converts the 'C' program, consisting of 'C' program statements only (no directives), to the machine code or object code. We cannot directly execute this object code as it needs to be linked with the library files. The linker does this job of linking the object code with the standard library files to get the executable code.

#INCLUDE: This is the most commonly used pre-processor directive in a 'C' program. '#include' directive is used for *file inclusion*. Mostly one may need to include the standard header files to the program as he might be using a standard library function declared in that particular header file. The file to be include need to be a standard header file always, but one can include the other program file defining different function or variable declarations but not containing function main().

STANDARD HEADER FILES: The standard header files like "string.h" or "stdio.h" contains only functions prototypes and never contains the function bodies or definitions. The standard functions are already implemented and compiled to form library files. These library files are linked to your program while creating the ".exe" files. The function prototype declare in the standard header files provide the declaration interface which helps compiler understand that it is functions and what type of arguments it can take and what type of arguments it can take and what type of value it can return.

Library function			
Function	^{Return} Type	Purpose	Include File
abs (i)	int	Return the absolute value of i.	stdlib.h
acos (d)	double	Return the cos inverse of d.	math.h
asin (d)	double	Return the sine inverse of d.	math.h
atan (d)	double	Return the tan inverse of d.	math.h
atan2 (d1, d2)	double	Return the tan inverse of d1/d2.	math.h
atof (s)	double	Convert string s to a double-precision quantity.	stdlib.h
atoi (s)	int	Convert string s to an integer.	stdlib.h
atol (s)	long	Convert string s to a long integer.	stdlib.h
calloc (u1, u2)	Void* (pointer)	Allocate memory for an array having u1 elements, each of length u2 bytes. Return a pointer to the beginning of the allocate space.	malloc.h, or stdlib.h
ceil (d)	double	Return a value rounded up to the next higher integer.	math.h
cos (d)	double	Return the cosine of d.	math.h
cosh (d)	double	Return the hyperbolic cosine of d.	math.h
exit (u)	void	Close all files and buffers, and terminate the program. (Value of u is assigned by function, to indicate termination status.)	stdlib.h

exp (d)	double	Raise e to the power d	math.h
fabs (d)	double	Return the absolute value of d.	math.h
fclose (f)	int	Close file f. Return 0 if successfully closed.	stdio.h
feof (f)	int	Determine if an end-of-file condition has been reached. If so, return a nonzero value; otherwise, return 0.	stdio.h
fgetc (f)	int	Enter a single character from file f.	stdio.h
fgets (s, i, f)	char* Pointer	Enter string s, containing i characters, from file f.	stdio.h
floor (d)	double	Return a value rounded down to the next lower integer.	math.h
fmod (d1,d2)	double	Return the remainder of d1/d2 (with same sign as d1).	math.h
fopen (s1,s2)	file* Pointer	Open a file named s1 of type s2. Return a pointer to the file.	stdio.h
fputc (c, f)	int	Send a single character to file f	stdio.h
fputs (s, f)	int	Send string s to file f.	stdio.h
free (p)	void	Free a block of allocated memory whose beginning is indicated by p.	malloc.h, or stdlib.h
getc (f)	int	Enter a single character from file f.	stdio.h
getchar ()	int	Enter a single character from the standard input device.	stdio.h
gets (s)	char*	Enter string s from the standard input device.	stdio.h
isalnum (c)	int	Determine if argument is alphanumeric. Return a nonzero value if true ; 0 otherwise.	ctype.h
isalpha (c)	int	Determine if argument is alphabetic. Return a nonzero value if true; 0 otherwise.	ctype.h
isascii (c)	int	Determine if argument is an ASCII character. Return a nonzero value if true; 0 otherwise.	ctype.h
isctrl (c)	int	Determine if argument is an ASCII control character. Return a nonzero value if true; 0 otherwise.	ctype.h
isdigit (c)	int	Determine if argument is a decimal digit. Return a nonzero value if true; 0 otherwise.	ctype.h
isgraph (c)	int	Determine if argument is a graphic ASCII character. Return a nonzero value if true; 0 otherwise.	ctype.h
islower (c)	int	Determine if argument is lowercase. Return a nonzero value if true; 0 otherwise.	ctype.h
isodigit (c)	int	Determine if argument is an octal digit. Return a nonzero value if true; 0 otherwise.	ctype.h
isprint (c)	int	Determine if argument is a printing ASCII character (hex 0x20 – 0x7e; octal 040 – 176). Return a nonzero value if true; 0 otherwise.	ctype.h
ispunct (c)	int	Determine if argument is a punctuation character. Return a nonzero value if true; 0 otherwise.	ctype.h
isspace (c)	int	Determine if argument is a whitespace character. Return a nonzero value if true; 0 otherwise.	ctype.h
isupper (c)	int	Determine if argument is uppercase. Return a nonzero value if true; 0 otherwise.	ctype.h
isxdigit (c)	int	Determine if argument is a hexadecimal digit. Return a nonzero value if true; 0 otherwise.	ctype.h
labs (l)	long int	Return the absolute value of l.	math.h
log (d)	double	Return the natural logarithm of d.	math.h
log 10 (d)	double	Return the logarithm (base 10) of d.	math.h
malloc (u)	void*	Allocate u bytes of memory. Return a pointer to the	malloc.h, or

		beginning of the allocated space.	stdlib.h
Pow (d1, d2)	double	Return d1 raised to the d2 power.	math.h
putc (c, f)	int	Send a single character to file f.	stdio.h
putchar (c)	int	Send a single character to the standard output device.	stdio.h
puts (s)	int	Send string s to the standard output device.	stdio.h
rand ()	int	Return a random positive integer.	stdlib.h
rewind (f)	void	Move the pointer to the beginning of file f.	stdio.h
sin (d)	double	Return the sine of d.	math.h
sinh (d)	double	Return the hyperbolic sine of d.	math.h
sqrt (d)	double	Return the square root of d.	math.h
strcmp (s1, s2)	int	Compare two string lexicographically. Return a negative value if s1 < s2; 0 if s1 and s2 are identical; and a positive value if s1 > s2.	string.h
strcmpi (s1, s2)	int	Compare two string lexicographically, without regard to case. Return a negative value if s1 > s2; 0 if s1 and s2 are identical; and a positive value if s1 > s2.	string.h
strcpy (s1, s2)	char*	Copy string s2 to string s1.	string.h
strlen (s)	int	Return the number of characters in string.	string.h
strset (s, c)	char*	Set all characters within s to c (excluding the terminating null character / 0).	string.h
tan (d)	double	Return the tangent of d.	math.h
tanh (d)	double	Return the hyperbolic tangent of d.	math.h
toascii (c)	int	Convert value of argument to ASCII.	ctype.h
tolower (c)	int	Convert letter to lowercase.	ctype.h, or stdlib.h
toupper (c)	int	Convert letter to uppercase.	ctype.h, or stdlib.h

c denotes a character type argument
 d denotes a double precision argument
 f denotes a file argument
 i denotes an integer argument
 l denotes a long integer argument
 p denotes a pointer argument
 s denotes a string argument
 u denotes a unsigned integer argument

Exercise 1

[a] Which of the following are invalid variable names & why?

i] #MEAN	ii] population in 2006	iii] team's victory	iv] _basic
v] group.	vi] basic_hra	vii] 422	

valid variable name

[b] Point out the errors:-

i] int = 314.562*150; lossy

ii] name = 'Ajay';

iii] 3.14*r*r*h = vol_of_cyl;

iv] k = (a*b)(c+(2.5a+b)(d+e);

v] int a = b = 3 = 4;

[c] Suppose that i, j, and k are integer variables whose value are 1, 2, and 3 respectively.

$$i=1, j=2, b=3$$

$$2+2 \neq 4 \neq 4$$

$$2+8 \neq 4$$

$$2+0=2$$

F.E SEM-II

STRUCTURED PROGRAMMING APPROACH

Expression	$i < j$	$(i+j) >= k$	$(j+k) > (i+5)$	$k! = 3$	$j == 2$
value	1	1	0	0	1

[d] Suppose that i and j are integer variables whose values are 5 and 7, and f and g are following point variables whose value are 5.5 and -3.25.

Expression	$i += 5$	$f -= g$	$j* = (i - 3)$	$f/ = 3$
Value	10	8.750000	14	1.833333

Find out put of the following program

C2-P7:-

```
#include<stdio.h>
int main()
{int a=1,b=2,c=3,d=4.75,x;
clrscr();
x=++a + b++ * ++c%d++;
printf("%d %d %d %d %d ",a,b,c,d,x);
getch(); return(0);
}
```

Precedence & Associability
① /*% L to R
② +- L to R

Output:-

2 3 4 5 2

Expt: $a=1, b=2, c=3, d=4.75$
a becomes 2 & b becomes 3 due to pre-increment
 $x = 2 + 2 * 4 \% 5$
then b becomes 4 & c becomes 4 due to post-increment
then $2 + 8 = 10$
then $10 \% 4 = 2$

C2-P8:-

```
#include<stdio.h>
int main()
{ int x=1,y=1,z,w;
clrscr();
z = x++ + x++ + --x + ++x - ++y - y++;
w = x++ + --x + ++x + x++ + y++ + y++;
printf("%d %d %d %d ",x,y,z,w);
getch();return(0);
}
```

Output:-

5 0 15

value changes with compilers.

C2-P9:-

```
#include<stdio.h>
int main()
{ int x=3,y=4,z,w;
clrscr();
z = x++ + ++x + x++ + y-- + ++y;
w = x++ + ++x + --x + ++x + y++ + y++;
printf("%d %d %d %d ",x,y,z,w);
getch();return(0);
}
```

Output:-

6 2 5 5

C2-P10:-

```
#include<stdio.h>
int main()
{ int i=0;
clrscr();
printf("%d %d %d %d",i,i++,++i,i--);
getch();return(0);
}
```

Output:-

1 0 0 0

Expt: for right to left
prints i so then $i--$ so $i=0$
and $++i$ so $i=1$ & print $i=0$
and $i++$ so $i=1$ & print $i=1$
and $i--$ so $i=0$ & print $i=0$

C2-P11:-

```
#include<stdio.h>
int main()
{ int i=0;
clrscr();
printf("%d %d %d %d %d",++i,i++,i--,++i,--i);
getch();return(0);
}
```

Output:-

1 1 0 0 1

C2-P12:-

Output:-


```
#include<stdio.h>
int main()
{int x=1;
clrscr();
printf("%d %d %d\n",x,(x=x+2),(x<<2));
x<<2;
printf("%d %d %d\n",++x,x++,++x);
getch();
return(0);
}
```

modifies orig value of x
does not modify orig value of x
all 3 modify orig value of x

3 3 4
From R to L
 $x \ll 2$ $1 \ll 4 = 4$ so print 4 but still $x=1$
 $x=1+2=3$ so print 3
again print 3
 $x \ll 2$ does not modify x . still $x=3$
 $++x$ prints $x=4$
 $x++$ prints $x=4$ then $x=5$
 $++x$ first $x=6$ then print $x=6$

C2-13:-

```
#include<stdio.h>
int main()
{ int a=2,b=3,ab=4;
int i;
int in = '2'*2;
char ch='c';
clrscr();
printf("%c %c\n",ch,(++ch));
printf("%d %d %d\n",a,a,++a);
printf("%d %d %d\n",b,b,++b);
printf("%d %d %d\n",ab,ab,++ab);
printf("%d %d %d\n",a,!!a);
getch();return(0);}
```

ascii value of '2' = 50
*in = 50*2 = 100*
'not' of non-zero num = 0
eg: !2 = 0
'not' of zero = 1
!0 = 1

Output:-

d d
3 3 3 - R to L
4 4 4
5 5 5
3 1

$a=3$ is non-zero value
 $\therefore !a = 0$
 $!!a = 1$
but still $a=3$

C2-P14:- Find output.

```
#include<stdio.h>
void main()
{ int x=10,y,z;
clrscr();
z=y=x;
y=--x; // y = y - (--x)
z=x--; // z = z - (x--)
x=--x-x--; // x = x - (--x)
printf("%d %d %d",x,y,z);
getch();
}
```

modifies orig value of x
 $y = 10 - 1 \Rightarrow y = 9$
 $z = 10 - 9 \Rightarrow z = 1$
 $x = 9 - 9 \Rightarrow x = 0$

\uparrow
value changes from compiler to compiler
($y=1, z=1$ always)

C2-P15:- WAP to swap two variables without using third variable. N.Bk.

C2-P16:- Find output.

```
#include<stdio.h>
int main()
{int x=1,y=5;
clrscr();
printf("%d",++(x+y));
getch();return(0);
}
```

error: (L value read)

Expl: We can use increment operator only on single variable, not an expression

C2-P17:- Find output.

```
#include<stdio.h>
void main()
{int a,b,c;
a=!(5+5<10);
b=(5+5==10) || (1+3==5);
c=5.10 || 10<20 && 3<5;
printf("a=%d b=%d c=%d",a,b,c);
getch();
}
```

o/p: a = 1 b = 1 c = 1

Expl: $a = !0 \Rightarrow a = 1$
 $b = 1 || 0 \Rightarrow b = 1$
 $c = 1 || 1 \&\& 1$ (as $1 \&\& 1 = 1$)
 $= 1$

C2-P18:- Use of some library function.


```

#include<stdio.h>
#include<math.h>
void main()
{ float v,y;
clrscr();
printf("ceil(3.4)= %lf\n",ceil(3.4));
printf("ceil(-3.4)= %lf\n",ceil(-3.4));

printf("Absolute value of (-3.1164)= %lf\n",fabs(-3.1416));

printf("floor(3.4)= %lf\n",floor(3.4));
printf("floor(-3.4)= %lf\n",floor(-3.4));

printf("abs(-3.4)= %d\n",abs(-3.4));
printf("fmod(3.4,1.0)= %lf\n",fmod(3.4,1.0));

printf("Enter no.for logarithmic value = ");
scanf("%f",&v);
printf("Answer = %f \n",log10(v) );
printf("Enter no.for exponent value = ");
scanf("%f",&y);
printf("Answer = %f ",exp(y) );

getch();
}

```

Output :

ceil(3.4) = 4.000000
 ceil(-3.4) = -3.000000
 Absolute value of (-3.1416) = 3.141600
 floor(3.4) = 3.000000
 floor(-3.4) = -4.000000
 abs(-3.4) = 3
 fmod(3.4,1.0) = 0.400000
 Enter no. for logarithmic value = 2
 Answer = 0.301030
 Enter no. for exponent value = 2
 Answer = 7.389056

Syllabus topics:- Introduction, Definition and uses of Pointers, Address Operator , Pointer variables , Dereferencing Pointer, Void Pointer, Pointer Arithmetic, Pointers to Pointers

2.30 INTRODUCTION TO POINTERS

A pointer is a derived data type in C. It is built from one of the fundamental data types available in C. Pointers contain memory address as their values. Since these memory addresses are the locations in the computer memory when program instructions and data are stored, pointers can be used to access and manipulate data stored in memory.

2.31 ACCESSING A VARIABLE THROUGH ITS POINTER

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

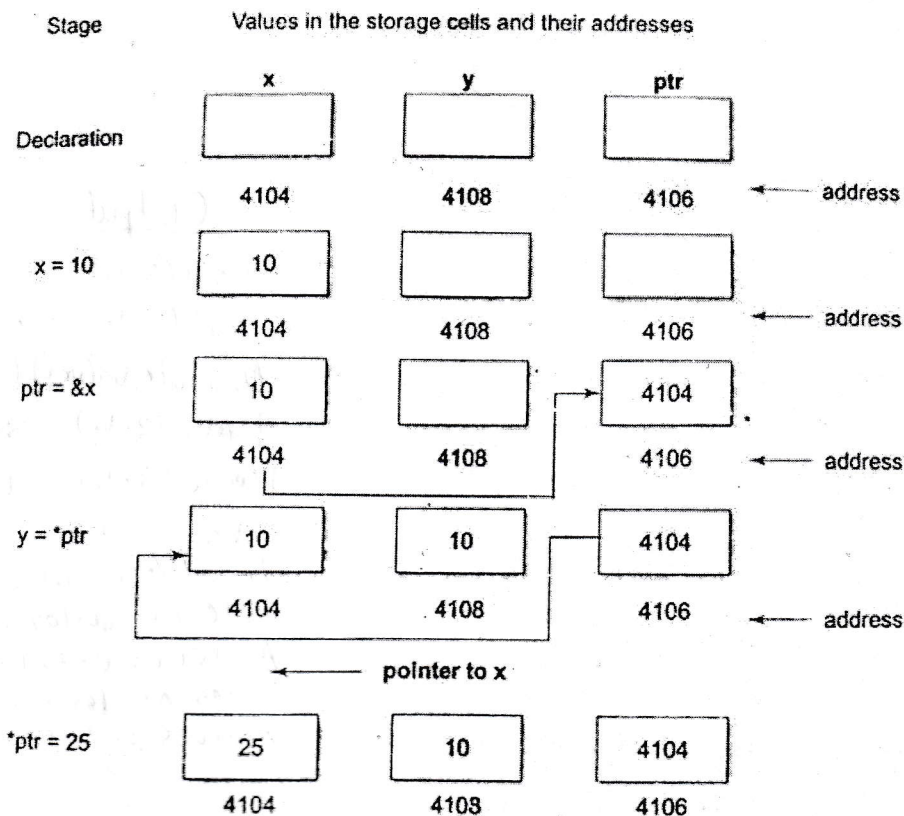
```

int quantity, *p, n;
quantity = 179;
p = &quantity;
n = *p;

```

The first line declares **quantity** and **n** as integer variables and **p** as a pointer variable pointing to an integer. The second line assigns the value 179 to **quantity** and the third line assigns the address of **quantity** to the pointer variable **p**. The fourth line contains the indirection operator *. When the operator * is placed before a pointer variable in an

expression (on the right-hand side of the equal sign), the pointer returns the value of the variable of which the pointer value is the address. In this case, ***p** returns the value of the variable **quantity**, because **p** is the address of **quantity**. The ***** can be remembered as 'value at address'. Thus the value of **n** would be 179.



C2-P19:--Referencing and Dereferencing

```
#include<stdio.h>
void main( )
{
    int quantity,*p,n;
    quantity = 179;
    p = &quantity; //referancing
    n=*p;          //de-referancing
    printf("value of quantity = %d ",*p);
    printf("address of quantity =%ld",p);
    getch();
}
```

2.33 POINTER INCREMENTS AND SCALE FACTOR

An expression like **p1++**; will cause the pointer **p1** to point to the next value of its type. For example, if **p1** is an integer pointer with an initial value, say 2800, then after the operation **p1 = p1 + 1**, the value of **p1** will be 2802, and not 2801. That is, when we increment a pointer, its value is increased by the 'length' of the data type that it points to. This length called the **scale factor**. The number of bytes used to store various data types depends on the system and can be found by making use of the **sizeof** operator. For example, if **x** is a variable, then **sizeof(x)** returns the number of bytes needed for the

variable. (Systems like Pentium use 4 bytes for storing integers and 2 bytes for short integers.)

<p>C2-P20:- find output.</p> <pre>#include<stdio.h> #include<conio.h> void main() { int x=20,y,*ip; clrscr(); ip=&x; y=(*ip)++; printf("%d\n",y); printf("%d\n",*ip); y=++(*ip); printf("%d\n",y); printf("%d\n",ip); getch(); }</pre>	<p>Output:-</p> <p>20 21 22 569261259</p>
<p>C2-P21:-</p> <pre>#include<stdio.h> int main() {int a,*p,**p1; clrscr(); a=125; p=&a; p1=&p; printf("%d\n",a); printf("%x\n",p); printf("%x\n",p1); printf("%d\n",*p); printf("%x\n",*p1); printf("%d\n",**p1); getch();return(0); }</pre>	<p>a 125</p> <p>p = add of a 6ff6</p> <p>p1 = add of p 6ff6</p> <p>value at p = a 125</p> <p>value at p1 = add of p 6ff6</p> <p>value at **p1 = value at p 125</p>
<p>C2-P22:-</p> <pre>#include<stdio.h> int main() { int a,*a1; float b,*b1; clrscr(); a1=&a; b1=&b; printf("%x \n %x \n",a1,b1); a1++; b1++; printf("%x \n %x \n",a1,b1); getch();return(0); }</pre>	<p>a1 = 6ff6</p> <p>b1 = 6ff0</p> <p>a1 = 6ff6</p> <p>b1 = 6ff6</p>

2.34 Void Pointer

A void pointer is a special type of pointer. It can point to any data type, from an integer value or a float to a string of characters.

Its sole limitation is that the pointed data cannot be referenced directly (the asterisk * operator cannot be used on them) since its length is always undetermined. Therefore,

type casting or assignment must be used to turn the void pointer to a pointer of a concrete data type to which we can refer.

```
C2-P23:-
#include<stdio.h>
void main()
{ int a=10;
  char b ='A';
  void *ptr=&a;

  //printf("%d",*ptr); (Error,void pointer can not be dereferance)

  printf("%d \n",*(int *)ptr);

  ptr=&b;

  //printf("%c",*ptr); (Error,void pointer can not be dereferance)

  printf("%c",*(char *)ptr);
  getch();
}
```


CH3. Expressing Algorithms - Selection

Many a times, we want a set of instructions to be executed in one situation, and an entirely different set of instructions to be executed in another situation. This kind of situation is dealt in C programs using a decision control instruction. As mentioned earlier, a decision control instruction can be implemented in C using:

(i) The if statement, (ii) The if-else statement, (iii) The conditional operator

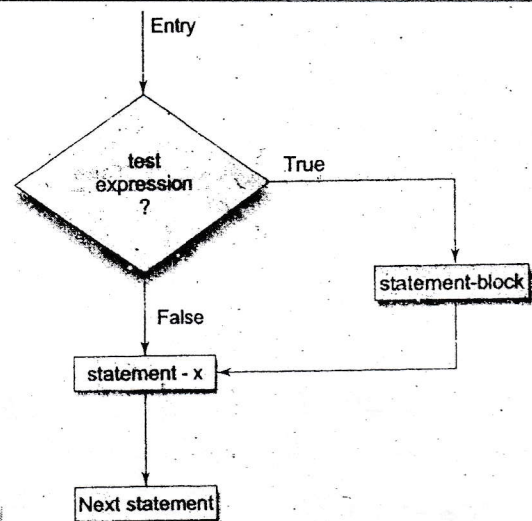
3.1 The 'if' statement:-

Syntax: if (this condition is true)
execute this statement;

C3-P1:- While purchasing certain items, a discount of 10% is offered if the quantity purchased is more than 1000. If quantity and price per item are input through the keyword, write a program to calculate the total expenses.

/* CALCULATION OF TOTAL EXPENSES*/

```
#include<stdio.h>
int main()
{ int qty;
float rate,tot,disc,discount,bill;
clrscr();
printf("Enter the quantity & rate: ");
scanf("%d %f",&qty,&rate);
if(qty>1000)
    disc=10;
    else
disc=0;
tot = qty*rate;
discount = qty*rate*disc/100;
bill = tot-disc;
printf("Total : %f Discount : %f Final
bill: %f",tot,discount,bill);
getch();
return(0);
}
```



Flowchart of simple if Control

Output:-

C3-P2:- Write a C program to display the following two prompts:-

Enter a month: (Use a 1 for Jan. etc.)

Enter a day of the month: Have your program accept and store a number in the variable month in response to the first prompt, and in response to the second prompt, accept and store a number in the variable day. If the month entered is not between 1 and 12, print a message inferring the user that an invalid month has been entered. If the day entered is not between 1 and 31. Print a message inferring the user that an invalid day has been entered.

```
#include<stdio.h>
int main()
{ int month,day;
clrscr();
printf("Enter month by no. (1:jan,2:feb...):");
```


F.E SEM-II STRUCTURED PROGRAMMING APPROACH

```
scanf("%d",&month);
printf("Enter a day of the month ");
scanf("%d",&day);
    if( month<1 || month>12 )
printf("Invalid month\n");
    else
printf("valid month\n");
    if( day<1 || day>31 )
printf("Invalid day\n");
    else
printf("valid day");
getch();return(0);
}
```

Output:-

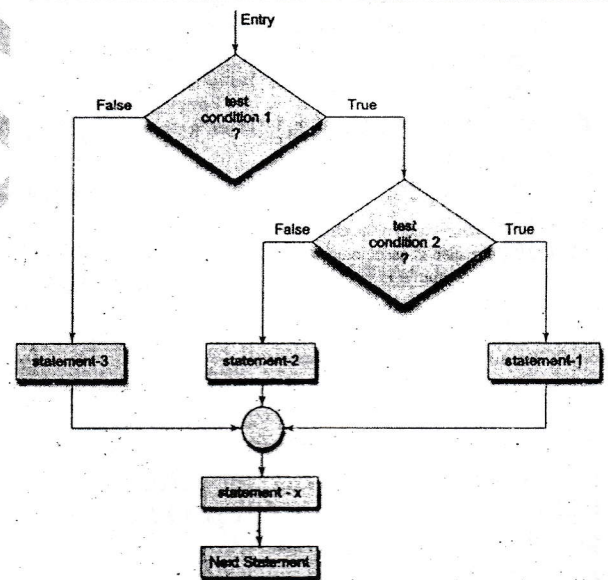
The group of statements after the 'if' and not including the 'else' is called an 'if block'.
Similarly, the statements after the else form the 'else block'.

3.2 Nested if-else:- It is perfectly all right if we write an entire if-else construct within the body of the 'if' statement or the body of an 'else' statement. This is called 'nesting' of if's.

C3-P3:-

```
#include<stdio.h>
int main()
{   int i;
    clrscr();
    printf("Enter 1 or 2: ");
    scanf("%d",&i);
    if ( i==1 )
        printf("E.D");
    else
    {
        if (i==2)
            printf("S.P.A");
        else
            printf("Maths2");
    }
    getch();return(0);
}
```

Output:-



Flow chart of nested if...else statements

Forms of if:-

<pre>(a) if (condition) do this ; (b) if (condition) { do this ; and this ; } (c) if (condition) do this ; else do this ; (d) if (condition) { do this ; and this ; } else { do this ; and this ; }</pre>	<pre>(e) if (condition) do this ; else { if (condition) do this ; else { do this ; and this ; } } (f) if (condition) { if (condition) do this ; else { do this ; and this ; } } else do this ;</pre>
---	---

3.3 Use of Logical Operators:

C allows usage of three logical operators, namely - &&, || and !. These are to be read as 'and' 'OR' and 'NOT' respectively.

C3-P4:-

The marks obtained by a student in 5 different subjects are input through the keyword.

The student gets a division as per the following rules:-

Percentage above or equal to 60 - First division

Percentage between 50 and 59 - Second division

Percentage between 40 and 49 - Third division

Percentage less than 40 - Fail.

Write a program to calculate the division obtained by the student.

```
#include<stdio.h>
int main()
{
    int m1,m2,m3,m4,m5,per;
    clrscr();
    printf("Enter marks in 5 sub.(out of 100) : ");
    scanf("%d %d %d %d %d",&m1,&m2,&m3,&m4,&m5);
    per = (m1+m2+m3+m4+m5)/5;
    if ( per>=60 )
        printf("1st division");
    if ( (per>= 50) && (per<60) )
        printf("2nd division");
    if ( (per>=40) && (per<50) )
        printf("3rd division");
    if ( per<40 )
        printf("Fail");
    getch();return (0);
}
```


F.E SEM-II STRUCTURED PROGRAMMING APPROACH

Output:-

- [a] The matching of the if's with their corresponding else's gets avoided, since there are no else's in the program.
- [b] In spite of using several conditions, the program doesn't creep to the right.

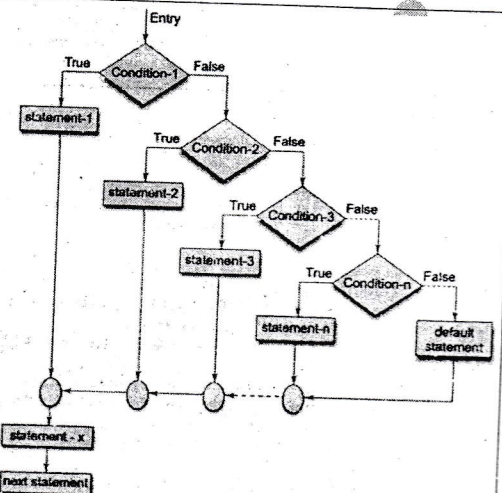
3.4 The else if clause:-

Note that the else if clause is nothing different. It is just a way of rearranging the else with the 'if' that follow it.

C3-P5:-

```
#include <stdio.h>
int main()
{ int score;
  clrscr();
  printf("Enter your test score: \n");
  scanf("%d",&score);
  if (score >= 90 )
    printf("Your grade is an A.");
  else if (score >= 80)
    printf("Your grade is a B.");
  else if (score >= 70)
    printf("Your grade is a C.");
  else if (score >= 60)
    printf("Your grade is a D.");
  else if (score >= 0)
    printf("Your grade is an F.");
  else
    printf("Invalid score");
  getch();return(0);
}
```

Output:-



Flow chart of else-if ladder

3.5 The Conditional Operator:-

The Conditional Operators ? and : are sometimes called ternary operator since they take three arguments

expression 1 ? expression 2 : expression 3 ;

What this expression says is : " if expression 1 is true (that is, if its value is non- zero), then the value returned will be expression 2; otherwise the value returned will be expression 3".

[a] int x, y;

scanf("%d",&x);

y = (x > 5 ? 3:4);

This statement will store 3 in y if x is greater than 5, otherwise it will store 4 in y.

3.6 THE GOTO STATEMENT:

A goto statement causes the program control to be transferred from one point

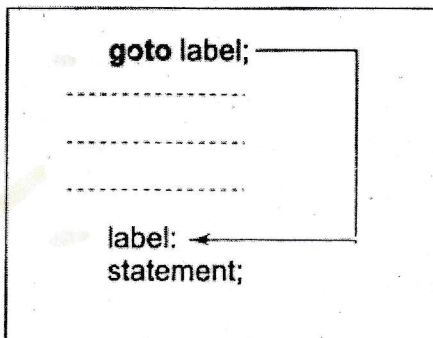
F.E SEM-II STRUCTURED PROGRAMMING APPROACH

to another. A goto statement transfers the control to a specific statement. This specific statement is identified by label which is an identifier followed by a colon. The syntax for the go to statement is,

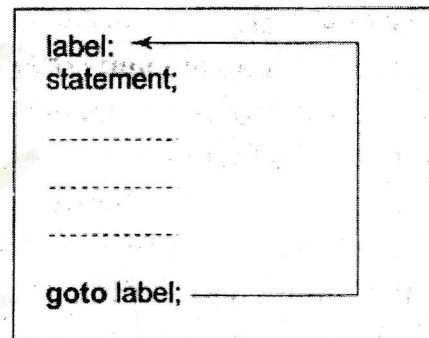
goto label ;

This label is not declared in the program. But it is attached to the statement with a colon, to which the goto statement will jump to. So, we can say that the goto statement jumps to a labeled statement with the same label.

This jump by the goto statement can be either forward or backward.



Forward jump



Backward jump

C3-P6:-

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

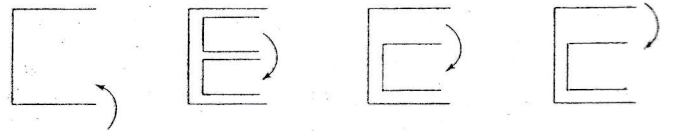
    printf("Do you want to continue (Y or N) ? : ") ;
    ans = getch() ;

    if ((ans == 'y') || (ans == 'Y'))
        goto again ;
    else
        printf("Thank You !!\n") ;
        getch();return(0) ;
}
```

Output:-

F.E SEM-II STRUCTURED PROGRAMMING APPROACH

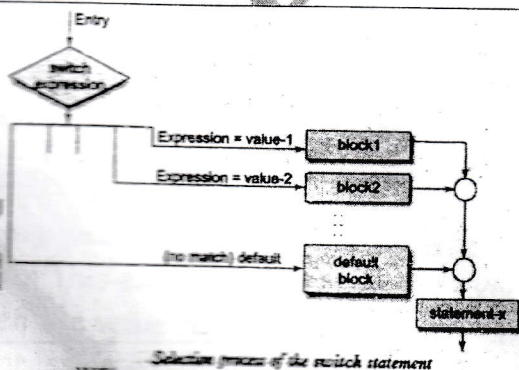
Avoiding goto:- It is a good practice to avoid using goto. When goto is used, many compilers generate a less efficient code. In addition, using many of them makes a program logic complicated and renders the program unreadable. The following goto jumps would cause problems and therefore must be avoided.



3.7 THE SIMPLE SWITCH:

The control statement which allows us to make a decision from the member of choice is called a *switch*, or more correctly a *switch-case-default*, since these three keywords go together to make up the control statement. They appear as follows:

```
Switch (integer  
expression)  
{  
    Case constant1 :  
do this  
break;  
    Case constant2 :  
do this  
break;  
    Case constant3:  
do this  
break;  
    default: do  
this;
```



When there is a switch statement, it evaluates the *expression* and then looks for a matching case label. If none is found, the *default* label is used. If no default is found, the statement does nothing. The default case, if present, will be selected if none of the prior cases are chosen. A default case is not required but it is good programming practice to include one.

The biggest defect in the switch statement is that cases do not break automatically after the execution of the corresponding statement list under a case is executed, the flow of control continues down, executing all the following cases until a break statement is reached.

When the break statement is executed within a switch, C executes the next statement outside the switch construct.

side note
→ default is not given sp. status.
It is executed in line if its occurrence in the cases.
∴ 'break' is very imp

C3-P7:- Write a program to display months in word. (Month number is the input)

```
#include<stdio.h>  
int main()  
{  
    int n=0;  
    clrscr();  
    printf("Enter the Month Number(1-12) : \n");  
    scanf("%d",&n);
```


F.E SEM-II STRUCTURED PROGRAMMING APPROACH

```

switch(n)
{
    case 1:
        printf("Month is Jan");
        break;
    case 2:
        printf("Month is Feb");
        break;
    case 3:
        printf("Month is Mar");
        break;
    case 4:
        printf("Month is Apr");
        break;
    case 5:
        printf("Month is May");
        break;
    case 6:
        printf("Month is June");
        break;
    case 7:
        printf("Month is July");
        break;
    case 8:
        printf("Month is Aug");
        break;
    case 9:
        printf("Month is Sep");
        break;
    case 10:
        printf("Month is Oct");
        break;
    case 11:
        printf("Month is Nov");
        break;
    case 12:
        printf("month is Dec");
        break;
    default:
        printf("Wrong input!!!");
}
getch(); return (0);
}

```

3.8 Compare the switch() control statement with control statement if-else-if.

No.	Switch-case	if-else
1	Case control structure	Decision control structure
2	Syntax: switch (integer expression) { case constant1: do this break; case constant 2: do this; break; case constant n:	Syntax: If (condition) { //do this } else { //do this }

F.E SEM-II STRUCTURED PROGRAMMING APPROACH

	<pre>do this; break; Default: do this; }</pre>	
3	<p>The switch statement has more than two options or branches. These options, in the switch statement are called as cases. So, we can have 'n' cases in a switch, with the last always being a default case. Whenever no value matches with the cases available, the default case is evaluated. But, to have the default case or no is optional. If the default case is not present, then when no value matches with either of the cases, the program simply quits the switch statement. We can call it as an advanced step of the if-else statement.</p>	<p>In the if-else statement, the 'if' part is executed if the expression evaluates to be TRUE and the else-part is executed if the expression evaluates to be FALSE. In other words, we can say that the if-else has only two options available.</p>
4	<p>Each case can have more than one statements included within. But, for the case statement, placing of these brackets is not a must. This is because; the case statement is a labeled statement. The use of brackets is optional.</p>	<p>If block of else block can also more than one statement. When there is more than one statement, placing of curly braces for the block is a must.</p>

C3-P8 :- WAP to find whether the first number entered by the user is divisible by the second number or not.

C3-P9 :- WAP to accept three sides of triangle from user and find the type of triangle.

C3-P10 :- WAP to find the minimum and maximum numbers out of the three numbers entered by the user by nested if-else statement.

C3-P11 :- WAP to check whether entered character is lower case, upper case, numeric or symbol using if-else only.

C3-P12 :- WAP to simulate a simple calculator using switch.

~~C3-P13 :- WAP to find whether the enter character is vowel or not by using switch.~~

C3-P14 :- Find output

```
void main()
{
    int choice=3;
    clrscr();
    switch(choice)
    {default:
        printf("mumbai ");
        case 1:printf("pune ");
        break;
        case 2:printf("nasik ");
        break;
    }
    getch();}
```

o/p: mumbai pune

(default pos" isn't fixed)

& break is missing

C3-P15 :- Print "Hello World" without using semicolon.

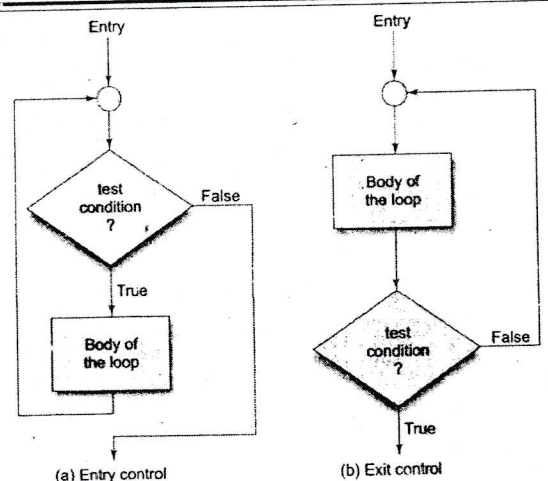
C3-P16 :- WAP to find whether the enter year is a leap year or not.

CH4. Expressing Algorithms - Iteration

Iteration is the repetition of statement of block of statements in a program. C has three iteration statements:

1. While statement,
2. Do...while statement,
3. For statement.

Iteration statements are also loops because of their cyclic nature.



Loop control structures

4.1 The while statement:-

The statement within the while loops would keep on getting executed till the condition being tested remains true. When the condition becomes false, the control passes to the first statement that follows the body of the while loop.

In place of the condition there can be any other valid expression.

```
while (test condition)
{
    body of the loop
}
```

So as long as the expression evaluated to a non-zero value the statement within the loop would get executed. The condition being tested may be relational or logical operators as shown in the following examples:-

```
while (i <= 10)
while (i >= 10 && j <= 15)
while (j > 10 && (b < 15 || c < 20))
```

It is not necessary that a loop counter must only be an int. It can even be a float.

4.2 The do-while Loop:-

do-while would execute its statements at least once, even if the condition fails for the first time. The while, on the other hand will not execute its statements if the condition fails for the first time.

```
do
{
    body of the loop
}
while (test-condition);
```

4.3 The for Loop:

The for statement allows us to specify three things about a loop in a single line:

- a) Setting a loop counter to an initial value.
- b) Testing the loop counter to determine whether its value has reaches the number of repetitions desired.
- c) Increasing the value of loop counter each time the program segment within the loop has been executed.

The syntax for the for statement is,

For (initialization; condition; update)

statement;

where initialization, condition and expressions are optional expression, statement is any

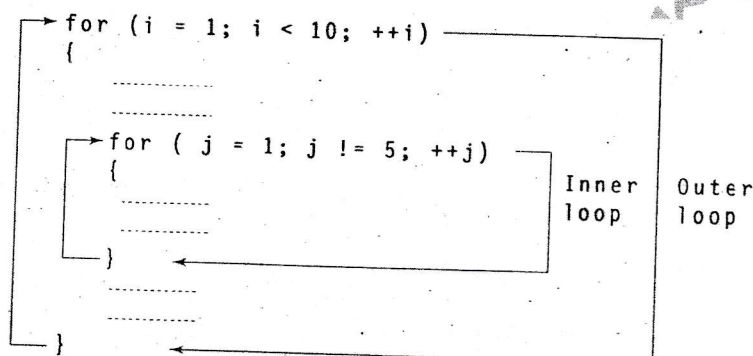
executable statement. The three parts (initialization; condition; update) control the loop.

The initialization expression is used to declare and/ or initialize control variables for the loop; it is evaluated first before any iteration occurs. We can have more than 1 expression separated by comma.

The condition expression is used to determine whether the loop should continue iterating, it is evaluated immediately after the initialization, if it is true, the statement is executed otherwise the loop is terminated. We can have more than one condition separated by comma.

The update expression is used to update the control variable; it is evaluated after the statement is executed. So the sequence of events that generate the iteration are:

1. Execute the initialization expression;
2. If the value of the condition expression is false, terminate the loop;
3. Execute the statement;
4. Execute the update expression;
5. Repeat steps 2-4.



4.4 Comparison of three loops:-

for
 for (n=1; n<=10; ++n)
 {

 }

while
 n = 1;
 while (n<=10)
 {

 n = n+1;
 }

do
 n = 1;
 do
 {

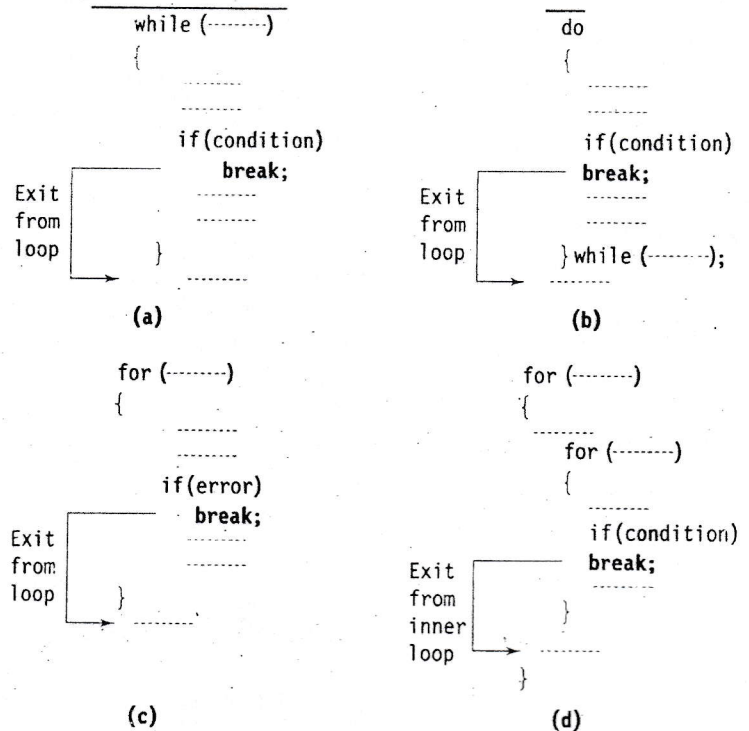
 n = n+1;
 }
 while(n<=10)

4.5 The Break

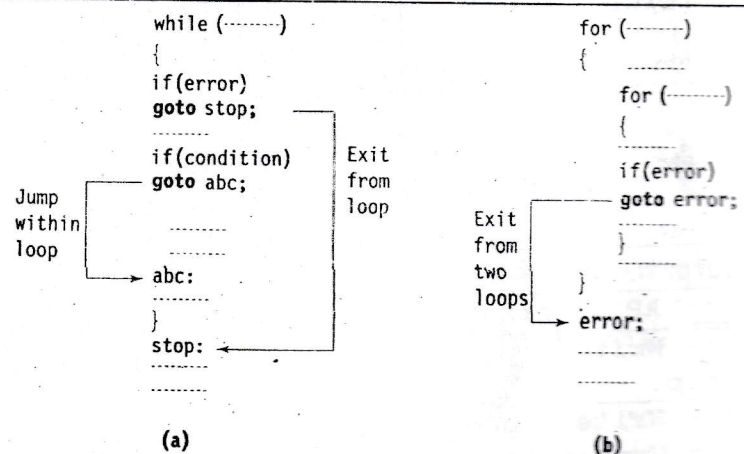
Statement:

When the keyword 'break' is encountered inside any loop, control automatically passes to the first statement after the loop. A break is usually associated with an 'if'.

If the break statement is used in the set of nested loops, then only the inner loop in which the break statement exists is terminated.



Exiting a loop with break statement

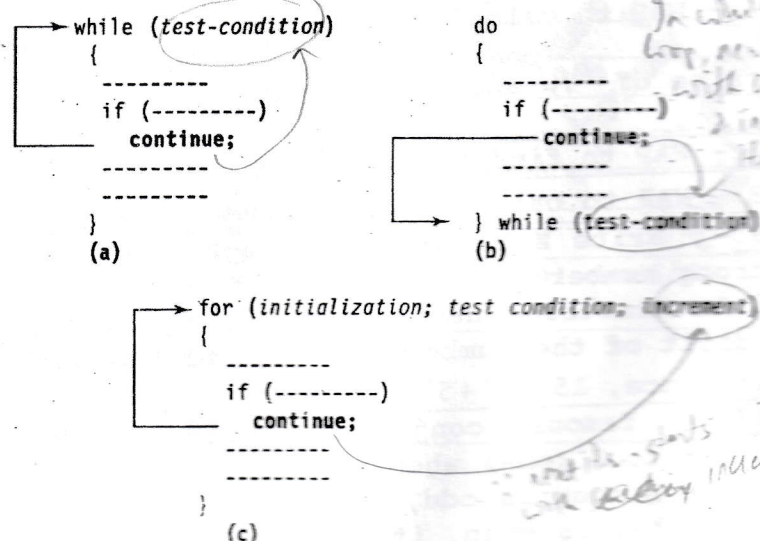


Jumping within and exiting from the loops with goto statement

4.6 The Continue

Statement:

In some programming situations we want to take the control to the beginning of the loop, bypassing the statement inside the loop which has not been executed. The keyword 'continue' allows us to do this. When the keyword continue is encountered inside any C loop, control automatically passes to the beginning of the loop. A 'continue' is usually associated with an 'if'.



Bypassing and continuing in loops

for(int i=1;i<=10;i++) { printf("A"); }	int i=1; for(;i<=10;) { printf("A"); i++; }	int i=1; while(i<=10) { printf("A"); i++; }
int i=1; do { printf("A"); i++; } while(i<=10);	for(int i=1;i<=10;i++) { for(int j=1;j<=5;j++) { printf("A"); } }	for(int i=1;i<=10;i++) { for(int j=1;j<=5;j++) { printf("A"); if(j==3) break; } }
for(int i=1;i<=10;i++) { for(int j=1;j<=5;j++) { if(j==3) break; printf("A"); } }	for(int i=1;i<=10;i++) { if(i==3) break; for(int j=1;j<=5;j++) { printf("A"); } }	for(int i=1;i<=10;i++) { for(int j=1;j<=5;j++) { printf("A"); } if(i==3) break; }

Find output:-

C4-P1:-WAP to print Multiplication tables of 1 to 10.

C4-P2:- Write a program to generate all combinations of 1,2,3, using for loop.

C4-P3:- Write a program to display whether a number is prime or not. A prime Number is one which is divisible by 1& itself.

C4-P5:- WAP to calculate factorial of given number using for loop.

C4-P6:- WAP to calculate nP_r and nC_r

The value of nP_r and nC_r is given as, ${}^nP_r = \frac{n!}{(n-r)!}$ and ${}^nC_r = \frac{n!}{r!(n-r)!}$

C4-P7:- WAP to find Sum of digits of an integer.

C4-P8:- WAP to calculate X^n without using pow() function.

C4-P9:- Write a program to check whether the entered number is an Armstrong number.

Hint: An Armstrong number is one, for which the sum of the cubes of each digit of the number is equal to the number itself.

For instance, $153 = 1^3 + 5^3 + 3^3$

C4-P13:-A famous conjecture holds that all positive integers converges to 1 (one) when treated in the following fashion.

1. If the number is odd, it is multiplied by three and one is added.
2. If the number is even, it is divided by two
3. Continuously apply above operations to the intermediate results

until the number converges to one.

Write a program to read an integer number from keyboard and implement the above mentioned algorithm and display all the intermediate values until the number converges to 1. Also count and display the number of iterations require for the convergence.

C4-P14:- WAP to illustrate the study of approximate level of intelligence of a person using formula $i=2+(y+0.5x)$, produce a table of values of i,y,x, where y varies from 1 to 2 and for each value of y,x varies from 5.5 to 12.5 in steps of 0.5.

C4-P16:- WAP to find whether entered integer is a palindrome or not.

C4-P16:- WAP to print entered integer number in words.

C4-P17:- WAP to accept x and n from user & calculate $x^0 + x^2 + x^4 \dots + x^n$

C4-P18:- WAP to calculate $1^1 + 2^2 + 3^3 \dots + n^n$

C4-P19:- WAP to evaluate the series $\frac{1}{2} + \frac{3}{4} + \frac{5}{6} + \frac{7}{8} \dots \dots \dots n$

C4-P20:- WAP to evaluate the series $\frac{1}{2} - \frac{3}{4} + \frac{5}{6} - \frac{7}{8} + \frac{9}{10} \dots \dots \dots n$

C4-P27:- Write a function to find GCD and LCM of two integer using Euclid's algorithm.

C4-P28:- Find output

```
#include <stdio.h>
int main()
{int i = 1, j = 1;
clrscr();
for (;;)
{
if (i>3)
break;
else j+=i;
printf("%d \n",j);
i+= j;
}
getch();return(0);
}
```

C4-P29:- find output

```
#include<stdio.h>
int main()
{int i;
for(i=0;i<8;i++)
{ if (i%2==0)
printf("%d\n",i+1);
else if (i%3==0)
continue;
else if (i%5==0)
break;
printf("\n End of Program \n");
}
printf("\n End of program \n");
getch();return(0);}
```

Practice Programs

C4-P4:- WAP to print all prime numbers from 1 to 100.

C4-P10:- WAP to print all armstrong numbers between 1 to 1000.

C4-P11:- WAP To find roots of quadratic equation.

C4-P12:- WAP to print first 10 Pythagoras triplet.

C4-P15:- WAP to reverse a integer number.

C4-P21:- Write program to evaluate $\sin x = x - x^3/3! + x^5/5! -$

C4-P22:- Write program to evaluate $\cos x = 1 - x^2/2! + x^4/4! - x^6/6!$
(PP)

C4-P23:- A program to evaluate the power series

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots + \frac{x^n}{n!}, 0 < x < 1$$

C4-P24:- A program to evaluate the series

$$\frac{1}{1-x} = 1 + x + x^2 + x^3 \dots + x^n$$

C4-P25:- WAP to find four digit perfect square numbers in which first two digits & last two digit are also perfect square.

C4-P26:-A positive integer is entered through the keyboard. Write function to obtain the prime factors of this number.

Programs of pattern

C4-P30:-

```
#include<stdio.h>
void main()
{ int i,j,n;
  clrscr();
  printf("Enter no:");
  scanf("%d",&n);
  for (i=1; i<=n; i++)
  { for (j=1; j<=i; j++)
    { printf("%2d",i); }
  }
  printf("\n");
  getch();
}
```

Enter No: 4

	1					
	2		2			
	3		3		3	
	4		4		4	4

C4-P31:-

```
#include<stdio.h>
void main()
{ int i,j,n;
  clrscr();
  printf("Enter no:");
  scanf("%d",&n);
  for (i=n; i>=1; i--)
  { for (j=n; j>=i; j--)
    { printf("%2d",i); }
  }
  printf("\n");
  getch();
}
```

Enter No: 4

	4					
	3		3			
	2		2		2	
	1		1		1	1

C4-P32:-

```
#include<stdio.h>
void main()
{int i,j,m=1,n;
  clrscr();
  printf("Enter no of rows =");
  scanf("%d",&n);
  for(i=1; i<=n; i++)
  { for(j=1;j<=i;j++)
    { printf("%3d",m);
      m++;
    }
  }
  printf("\n");
  getch();
}
```

Enter nO of rows = 4

	1						
	2		3				
	4		5		6		
	7		8		9	1	0

C4-P33:-

```
#include<stdio.h>
void main()
{ int i,j,n;
  clrscr();
  printf("Enter no of row:");
  scanf("%d",&n);
  for (i=1; i<=n; i++)
  { if(i%2!=0) //odd row
    for (j=1; j<=i; j++)
    { printf("%d ",j); }
```

```
{ printf("%d ",j); }
printf("\n");
}
getch();
}
```

Enter no of row: 5

1							
2	1						
1	2	3					
4	3	2	1				


```
else //even row
    for (j=i; j>=1; j--)
```

1	2	3	4	5
---	---	---	---	---

C4-P34:-

```
#include<stdio.h>
void print(int,int);
void main()
{ int i,j,n,k,a;
  clrscr();
  printf("Enter no of row:");
  scanf("%d",&n);
```

```
    for (i=1; i<=n; i++)
        print(i,n);
  getch();
}
```

```
void print(int i,int n)
{ int j,k,a=0;
  k=n-1;
  for (j=1; j<=i; j++)
```

```
  {
    if(j==1)
```

```
printf("%3d",i);
    else if(j==2)
    { a=i;
      a=a+k;
      printf("%3d",a);

    }
    else
    { k--;
      a=a+k;
      printf("%3d",a);

    }
    printf("\n");
  }
```

Enter no. of Row:- 5

	1									
	2		6							
	3		7	1	0					
	4		8	1	1	1	3			
	5		9	1	2	1	4	1	5	

C4-P35:-

```
#include<stdio.h>
void main()
{int i,j,n;
  clrscr();
  printf("Enter no of rows=");
  scanf("%d",&n);
  for ( i=1;i<=n;i++)
```

```
{ for(j = 1;j<=i;j++)
  {printf("%2d",(i+j+1)%2 );
  }
  printf("\n");
}
getch();
}
```

Enter No of rows = 4

1					
0		1			
1		0		1	
0		1		0	1

C4-P36:-

```
#include<stdio.h>
void main()
{ int i,j,k,m,n;
  clrscr();
  printf("Enter no of rows:");
  scanf("%d",&n);
  for(i=1; i<=n; i++)
  {
    for ( k=1; k<=i; k++)
    { printf("%d ",k); }
    for ( m=i-1; m>=1; m--)
    { printf("%d ",m); }
    printf("\n");
  }
  for(i=n-1; i>=1; i--)
```

```
{
  for ( k=1; k<=i; k++)
  { printf("%d ",k); }
  for ( m=i-1; m>=1; m--)
  { printf("%d ",m); }
  printf("\n");
}
getch();
}
```

Enter No of rows = 4

1							
1	2	1					
1	2	3	2	1			
1	2	3	4	3	2	1	
1	2	3	2	1			
1	2	1					
1							

C4-P37:-

```
#include<stdio.h>
void main()
{ int i,j,k,m,n;
  clrscr();
  printf("Enter no of rows");
  scanf("%d",&n);
```

Output

```
Enter No of rows = 5
5
54
543
5432
```

```

for ( i=1; i<=n; i++)
{
    for ( j=1; j<=n-i; j++)
        printf(" ");
    for ( k=i; k>=1; k--)
        printf("%d",k);
    for ( m=1; m<=i-1; m++)
        printf("%c", (m+'A'-1));
    printf("\n");
}
getch();
}

```

54321

C4-P38:-

```

#include<stdio.h>
void main()
{ int i,j,k,m,x,n;
  clrscr();
  printf("Enter no of rows:");
  scanf("%d",&n);
  for(i=n; i>=1; i--)
  { for (j=1; j<=i; j++)
    { printf("%d",j); }
    for (k=1; k<=(2*(n-i)-1); k++)
      printf(" ");
    if (i==n)
      x = i-1;
    else x=i;
    for (m=x; m>=1; m--)
      { printf("%d",m); }
    printf("\n");
  }
  getch();
}

```

Output

Enter No of rows = 4

```

1   2   3   4   3   2   1
1   2   3           3   2   1
1   2               2   1
1                   1

```

C4-P39:-

```

#include<stdio.h>
void main()
{ int i,j,k,m,n;
  clrscr();
  printf("Enter no of rows:");
  scanf("%d",&n);
  for(i=1; i<=n; i++)
  { for(j=1; j<=2*(n-i); j++)
    { printf(" "); }
    for (k=1; k<=i; k++)
      { printf("%d",k); }
    for (m=i-1; m>=1; m--)
      { printf("%d",m); }
    printf("\n");
  }
  getch();
}

```

Enter No of rows = 4									
					1				
			1	2	1				
	1	2	3	2	1				
1	2	3	4	3	2	1			

C4-P40:-

```

#include<stdio.h>
void main()
{ int s=1,i,j,k,m;
  for(i=1; i<=4; i++)
  { for(j=4; j>i; j--)
    { printf(" "); }
    for(k=1; k<=i; k++)
      { printf("%d",s); }
    s++;
    s--;
    for(m=1; m<i; m++)
      { printf("%d",--s); }
    s++;
    printf("\n");
  }
  getch();
}

```

			1			
		2	3	2		
	3	4	5	4	3	
4	5	6	7	6	5	4

C4-P41:-

```

#include<stdio.h>
void main()
{int i,k,n,r,P,C,factorial,temp,result;
  clrscr();
}

```



```

for(n=0;n<=4;n++) /*logic for rows*/
{
    for(k=1;k<=4-n;k++) /*logic for blank spaces*/
    { printf(" ");}
    {
        for (i=1,factorial=1 ; i<=n ; i++) /* Calculate n! */
            { factorial = factorial * i ; }
        for(r=0;r<=n;r++)
        {
            for (i=1,temp=1 ; i<=r ; i++) /* Calculate r! */
                { temp=temp*i; }

            for (i=1,result=1 ; i<=(n-r) ; i++) /*Calculate (n-r)!*/
                {result = result * i ;}

            P = factorial / result ;
            C = P / temp ;

            printf(" %d",C);
        }
    }
    printf("\n");
}
getch();
}

```

				1			
			1		1		
		1		2		1	
	1		3		3		1
1		4		6		4	1

						3C_0			
				1C_0			1C_0		
			2C_0		2C_1			2C_2	
		3C_0		3C_1		3C_2			3C_3
	4C_0		4C_1		4C_2		4C_3		4C_4

C4-P42:-

```

#include<stdio.h>
#include<conio.h>
void main()
{ int m,n,x;
printf("Enter no of rows ");
scanf("%d",&x);
for(n=1;n<=x;n++)
{ for(m=x;m>0;m--)
{ if(m==n)
printf("*");
else
printf("%d",m);
}
printf("\n");
}
getch();
}

```

Output

Enter No of rows = 5

```

5432*
543*1
54*21
5*321
*4321

```

Write Programs to generate different patterns by using Loop Control Structures:

C4-P43:-

Enter no of rows: 4

*							
*		*					
*		*		*			
*		*		*		*	

C4-P44:-

Enter no of rows:4

*		*		*		*	
*		*		*			
*		*					
*							

C4-P45:-

Enter no of rows: 4

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

C4-P46:-

Enter no of rows: 4

			*
		*	*
	*	*	*
*	*	*	*

C4-P47:-

Enter no of rows: 4

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

C4-P48:-

Enter no of rows: 4

*	*	*	*
	*	*	*
		*	*
			*

C4-P49:-

Enter no of rows: 4

*		*		*		*	
		*		*		*	
				*		*	
						*	

C4-P50:-

Enter no of rows: 4

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

C4-P51:-

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

C4-P52:-

Enter no of row:5

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

C4-P53:-

Enter no of row:5

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

C4-P54:-

Enter no of rows: 4

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

C4-P55:-

Enter capital Alphabate:D							
A							
A		B					
A		B		C			
A		B		C		D	

C4-P56:-

Enter capital alphabate:D							
D							
C		C					
B		B		B			
A		A		A		A	

C4-P57:-

Enter capital alphabate:D							
A		A		A		A	
B		B		B			
C		C					
D							

C4-P58:-

Enter capital alphabate:D							
A		B		C		D	
		A		B		C	
				A		B	
						A	

C4-P59:-

Enter capital alphabate:D							
D		C		B		A	
		C		B		A	
				B		A	
						A	

C4-P60:-

Enter capital alphabate:D							
						A	
					A	B	
			A	B	C		
A	B	C	D				

C4-P61:-

Enter capital alphabate:D							
						D	
				D		C	
		D	C	B			
D	C	B	A				

C4-P62:-

				A
			C	B
		F	E	D
	J	I	H	G
O	N	M	L	K

C4-P63:-

Enter capital alphabate:D							
A	B	C	D	C	B	A	
A	B	C		C	B	A	
A	B				B	A	
A						A	

C4-P64:-

Enter no of rows:4											
				A							
			B			B					
	C			C			C				
D			D			D					D

C4-P65:-

Enter no of rows:4											
				A							
			A	B	A						
	A	B	C	B	A						
A	B	C	D	C	B	A					

C4-P66:-

Enter no of rows:4											
			1								
		2	1	A							
	3	2	1	A	B						
4	3	2	1	A	B	C					

<u>C4-P67:-</u> Enter No of rows = 4 4 43 432 4321	<u>C4-P68:-</u> Enter No of rows = 4 A BC DEF GHIJ																																			
<u>C4-P69:-</u> Enter No of rows = 4 A CB FED JIHG	<u>C4-P70:-</u> <table><tr><th colspan="7">Enter no of rows:4</th></tr><tr><td></td><td></td><td></td><td>1</td><td></td><td></td><td></td></tr><tr><td></td><td></td><td>1</td><td>2</td><td>A</td><td></td><td></td></tr><tr><td></td><td>1</td><td>2</td><td>3</td><td>A</td><td>B</td><td></td></tr><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>A</td><td>B</td><td>C</td></tr></table>	Enter no of rows:4										1						1	2	A				1	2	3	A	B		1	2	3	4	A	B	C
Enter no of rows:4																																				
			1																																	
		1	2	A																																
	1	2	3	A	B																															
1	2	3	4	A	B	C																														

- check answers id for soln.
- ① C4-P71 : WAP for strong nos. ($145 = 1! + 4! + 5!$)
- ② C4-P72 : WAP for strong nos (upto n terms)
- ③ C4-P73 : WAP for perfect nos ($1+2+3=6$)
sum of divisors = num
- ④ C4-P74 : ~~WAP~~ (upto n terms)

CH5. Decomposition of Solution - FUNCTIONS

5.1 Modular Programming

Modular programming is a strategy applied to the design and development of software systems. It is defined as organizing a large program into small, independent program segments called **modules** that are separately named and individually called program units. These modules are carefully integrated to become a software system that satisfied the system requirements. It is basically a "divide-and-conquer" approach to problem solving. Modules are identified and designed such that they can be organized into a top-down hierarchical structure. In C, each module refers to a function that is responsible for a single task.

Some characteristics of modular programming are:

1. Each module should do only one thing.
2. Communication between modules is allowed only by a calling module.
3. A module can be called by one and only one higher module.
4. No communication can take place directly between modules that do not have calling-called relationship.
5. All modules are designed as *single-entry, single exit* systems using control structures.

'C' functions are classified into two categories:-

- [1] Library functions.
- [2] User - defined functions.

• **Library Functions:** - 'C' functions which have predefined meaning to 'C' compiler are known as library function or standard functions. For e.g. `sqrt()`, `abs()`, `strcat()`, `cos()` etc..

• **User - Defined Functions:** - Apart from standard functions, if we define our own functions, they are called user-defined functions. In order to make use of a user-defined function, we need to establish three elements that are related to functions.

1. Function definition (for implementation)
2. Function call
3. Function declaration (for prototype)

The function definition is an independent program module that is specially written to implement the requirements of the function. In order to use this function we need to invoke it at a required place in the program. This is known as the **function call**. The program that calls the function is referred to as the **calling program or calling function**. The calling program should declare any function that is to be used in the program. This is known as the **function declaration or function prototype**.

5.2 DEFINITION OF FUNCTIONS

A **function definition**, also known as **function implementation** shall include the following elements.

1. Function name	4. Local variable declarations
2. Function type (return type)	5. Function statements
3. List of parameter	6. A return statement

All the six elements are grouped into two parts, namely,

- Function header (First three elements)
- Function body (Second three elements)

A general format of a function definition to implement these two parts is give below:

```
function_type function_name (parameter list)
{
    local variable declaration; executable statement1;
    executable statement2;
    return statement;
}
```

parameters declared here are called dummy or formal parameters

5.3 Function Body

The function body contains the declarations and statements necessary for performing the required task. The body enclosed in braces, contains three parts, in the order given below:

1. Local declarations that specify the variables needed by the function.
2. Function statements that perform the task of the function.
3. A return statement that returns the value evaluated by the function.

If a function does not return any value, we can omit the return statement. However, note that its return type should be specified as void.

1. When a function reaches its return statement, the control is transferred back to the calling program. In the absence of a return statement, the closing brace acts as a int return.
2. A local variable is a variable that is defined inside a function and used without having any role in the communication between functions.

5.4 FUNCTION CALLS

(parameters passed are called real/actual parameters)

A function can be called by simply using the function name followed by a list of actual parameters (or arguments), if any, enclosed in parentheses.

5.5 FUNCTION DECLARATION

A function declaration (also known as function prototype) consists of four parts.

- Function type (return type)
- Function name
- Parameter list
- Termination semicolon

parameters declared here are called dummy or formal parameters

They are coded in the following format:

Function-type function-name (parameter list);

Points to note:

1. The parameter list must be separated by commas.
2. The parameter names do not need to be the same in the prototype declaration and the function definition.
3. The types must match the types of parameters in the function definition, in number and order.
4. Use of parameter names in the declaration is optional.

5. If the function has no formal parameters, the list is written as (void).
6. The return type is optional, when the function returns int type data.
7. The return type must be void if no value is returned.
8. ~~We~~ When the declared types do not match with the types in the function definition, ^{at} the compiler will produce an error.

A prototype declaration may be placed ^{at} in two places in a program.

1. Above all the functions (including main)
2. Inside a function definition.

When we place the declaration above all the functions (in the global declaration section), the prototype is referred to as a global prototype. Such declarations are available for all the functions in the program.

When we place it in a function definition (in the local declaration section), the prototype is called a *local prototype*. Such declarations are primarily used by the functions containing them.

The place of declaration of a function defines a region in a program in which the function may be used by other functions. This region is known as the *scope* of the function.

It is a good programming style to declare prototypes in the global declaration section before **main**. It adds flexibility, provides an excellent quick reference to the functions used in the program, and enhances documentation.

Prototype declarations are not essential. If a function has not been declared before it is used, C will assume that its details available at the time of linking. Since the prototype is not available, C will assume that the return type is an integer and that types of parameters match the formal definitions. If these assumptions are wrong, the linker will fail and we will have to change the program. The moral is that we must always include prototype declarations, preferably in global declaration section.

Parameters Everywhere !!

1. In declaration (prototypes)
2. In function call
3. In function definition.

The parameters used in prototypes and function definitions are called formal parameters and those used in function calls are called actual parameters. Actual parameters used in a calling statement may be simple constants, variables or expressions.

The formal and actual parameters must match exactly in type, order and number. Their names, however, do not need to match.

5.6 CATEGORY OF FUNCTIONS

A function, depending on whether arguments are present or not and whether a value is returned or not, may belong to one of the following categories.

Category 1: Functions with no arguments and no return values.

Category 2: Functions with arguments and no return values.

Category 3: Functions with arguments and one return a value.

Category 4: Functions with no arguments but return a value.

Category 5: Functions that return multiple values. (i.e. ^{at} with multiple return statements)

- ⑦ when there is a conflict between a local and global variable, the local variable is given preference.

5.7 Storage classes in C:-

To fully define a variable one needs to mention not only its 'type' but also its 'storage class'. In other words, not only do all variables have a data type, they also have a 'storage class'. From C compiler's point of view, variable name identifies some physical location within the computer where the string of bits representing the variable's value is stored. There are basically two kinds of location in a computer where such a value may be kept - Memory and CPU registers. It is the variable's storage class that determines in which of these two locations the value is stored.

Moreover, a variable's storage class tells us:-

- [i] What will be the initial value of the variable, if initial value is not specifically assigned (i.e. the default initial value).
- [ii] What is the scope of the variable.
- [iii] What is the life of the variable.

Scope : The region of a program in which a variable is available for use.

Visibility : The program's ability to access a variable from the memory.

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

There are four storage classes in C:

- [a] Automatic storage class
- [b] Register storage class
- [c] Static storage class
- [d] External storage class

[a] Automatic storage class:-

The features of a variable defined to have an automatic storage class are as under:-
Storage - memory.

Default initial value - An unpredictable value, which is often called a garbage value.

Scope - Local to the block in which the variable is defined.

Life - Till the control remains within the block in which the variable is defined.

Following program show how an automatic storage class variable is declared, and the fact that if the variable is not initialized it contains a garbage value.

C5-P1:-

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

Output:

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


[b] Register storage class:- The features of a variable defined to be of register storage class one is as under:-

Storage - CPU registers.

Default initial value - Garbage value.

Scope - Local to the block in which the variable is defined.

Life - Till the control remains within the block in which the variable is defined.

A value stored in a CPU register can always be accessed faster than the one that is stored in memory. Therefore, if a variable is used at many places in a program it is better to declare its storage class as register. A good example of frequently used variables is loop counter. We can name their storage class as register.

C5-P2:-

```
#include<stdio.h>
int main()
{
    register int i;
    clrscr();
    for ( i=1; i<=5; i++ )
        printf("\ni= %d",i);
    getch();return(0);}
```

Output:

```
i=1
i=2
i=3
i=4
i=5
```

Not every type of variable can be stored in a CPU register. For example, if the microprocessor has 16-bit register then they cannot hold a float value or a double value, which require 4 and 8 bytes respectively. However, if you use the register class for a float or a double variable you won't get you any error messages. All that would happen is the compiler would treat the variables to be of auto storage class.

[c] Static Storage class:-

The features of a variable defined to have a static storage class are as under:-

Storage - Memory.

Default initial value - zero

Scope - Local to the block in which the variable is defined.

Life - value of the variable persists between different function calls.

The difference between the automatic and static storage classes is as shown below:-

C5-P3:-

```
#include<stdio.h>
int increment()
{ auto int i = 1;
  printf("i= %d \n",i);
  i++;
}
int main()
{ clrscr();
  increment();
  increment();
  increment();
  getch();return(0);}
```

o/p:

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

C5-P4:-

```
#include<stdio.h>
int increment()
{ static int i = 1;
  printf("i= %d \n",i);
  i++;
}
int main()
{ clrscr();
  increment();
  increment();
  increment();
  getch();return(0);}
```

o/p:

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

In the above example, when variable i is auto, each time increment() is called it is re-initialized to one. When the function terminates, i vanishes and its new value of 2 is lost. The result: no matter how many times we call increment(), i is initialized to 1 every time.

On other hand, if *i* is static, it is initialized to 1 only once, It is never initialized again the first call to increment (), *i* is incremented to 2. Because *i* is static, this value persists. The next time increment () is called, *i* is not re- initialized to 1; on the contrary its old value 2 is still available. This current value of *i* (i.e.2) gets printed.

[d] External Storage Class:-

The features of a variable whose storage class has been defined as external are as follows:-

Storage - Memory

Default initial value - zero

Scope - Global

Life - As long as the program's execution doesn't come to an end.

External variable differ from those we have already discussed. In that their scope is global, not local. External variable are declared outside all functions, yet are available to all functions that care to use them.

```
// include <...>
int a = 5; // automatically enters class
void main()
{
    int a = 10;
    printf("%d", a);
    getch();
}
```

resolution operator (::) → o/p: 10

5.8 Scope and Lifetime of Variables

Storage Class	Where declared	Visibility (Active)	Lifetime (Alive)
None	Before all functions in a file (may be initialized)	Entire file plus other files where variable is declared with extern	Entire program (Global)
Extern	Before all functions in a file (cannot be initialized)	Entire file plus other files where variable is declared extern and the file where originally declared as global.	Global
Static	Before all functions in a file	Only in that file	Global
None or auto	Inside a function (or a block)	Only in that function or block	Until end of function or block
Register	Inside a function or block	Only in that function or block ..	Until end of function or block
Static	Inside a function	Only in that function	Global

5.9 Rules of use

The scope of a global variable is the entire program file.

The scope of a local variable begins at point of declaration and ends at the end of the block or function in which it is declared.

The scope of a formal function argument is its own function.

The lifetime (or longevity) of an auto variable declared in main is the entire program execution time, although its scope is only the main function.

The life of an auto variable declared in a function ends when the function, its lifetime extends till the end of program execution.

A static local variable, although its scope is limited to its function, its lifetime extends

till the end of program execution.

All variables have visibility in their scope, provided they are not declared again.

If a variable is redeclared within its scope again, it loses its visibility in the scope of the redeclared variable. (obviously)

Which to use when:- We can make a few ground rules for usage of different storage classes in different programming situations with a view to:

[a] Economize the memory space consumed by the variables.

[b] Improve the speed of execution of the program.

The rules are as under:-

[i] Use static storage class only if you want the value of a variable to persist between different function calls.

[ii] Use register storage class for only those variables that are being used very often in a program.

[iii] Use extern storage class for only those variable that are being used by almost all the functions in the program. This would aint unnecessary passing of these variables as arguments when making a function call.

[iv] If you don't have any of the express needs mentioned above, then use the auto storage class.

5.10 Recursion: - When a called function in turn calls another function a process of chaining occurs. Recursion is the special case of this process, where the function calls itself.

```
#include <stdio.h>
long int factorial(int);
void main()
{int n;
 long int fact,ans;
 clrscr();
 printf("Enter no for factorial ");
 scanf("%d",&n);
 ans=factorial(n);
 printf("Ans= %ld",ans);
 getch();
}
long int factorial(int n)
{ long int fact;
  if(n==1 || n==0)
    return 1;
  else
  { fact=n*factorial(n-1);
    return (fact);
  }
}
```

Let us see how the recursion works. Assume $n=3$. Since the value of n is not 1, The statement `fact = n*factorial (n-1);`

will be executed, with $n = 3$ *factorial(2); and

will be evaluated. The expression on the right -hand side includes a call to factorial with $n=2$. This call will return the following value $2*factorial(1)$. Once again, factorial is called with $n=1$. This time the function returns 1. The sequence of operations can be summarized as follows: `fact = 3*factorial(2)`

$$= 3 * 2 * \text{factorial}(1)$$

$$= 3 * 2 * 1$$

$$= 6$$

Recursive function can be effectively used to solve problems where solution is expressed in terms of successively applying the same solution to subsets of the problem. When we write recursive functions, we must have an if statement somewhere to force the function to return without the recursive call being executed. Otherwise, the function will never return. (infinite loop)

C5-P5:-WAP to find the Factorial for a number by Recursive function.

(prev. pg.)

C5-P6:-Write a program to obtain the prime factors recursively.

NRL

C5-P7:-Write a recursive function to obtain the sum of first n natural numbers.

NRL

C5-P8:-Write a recursive function that accepts one positive integer and will calculate and return the sum of all the digits present in it. Write a suitable main function.

NRL

C5-P9:- WAP using recursive function 'power' to compute x^n

Power(x,n)=1
Power(x,n)=x
Power(x,n)=x*power(x, n-1)

if n=0
if n=1
otherwise

first read right

```
#include<stdio.h>
int power(int,int);
void main( )
{int x,n;
clrscr();
printf("Enter base no & power: ");
scanf("%d %d",&x,&n);
printf("Answer: %d",power(x,n) );
getch( );
}
int power(int x,int n)
{ if (n==0)
    return 1;

    else if(n==1)
        return x ;
    else
        return (x*power(x,n-1));
}
```


C5-P10:-Write a recursive function that accepts two positive integers and calculates and returns their GCD using Euclid's algorithm. Write suitable main function. The Euclid's algorithm to calculate GCD of two numbers is given as follows.

GCD (m,n) = GCD (n, m) if m<n
 = m if n=0
 = GCD (m, m%n) otherwise

} use logic to remember ~~code~~
 (in case they don't give it)

```
#include<stdio.h>
int gcd(int,int);
void main( )
{int x,y;
clrscr();
printf("Enter 2 positive integers : ");
scanf("%d %d",&x,&y);
printf("GCD: %d",gcd(x,y) );
getch( );
}
int gcd(int m,int n)
{ if (m<n)
    return gcd(n,m);

    else if(n==0)
        return m ;
    else
        return gcd(n,m%n);
}
```

C5-P11:-Write a program on the recursive function called as the Ackerman's function which is popular among the lecturers of computer science. It can be given as:

ACK(m,n) = n+1 if m=0
 = ACK (m-1,1) if n=0 and m>0
 = ACK (m-1, ACK (m, n-1)) otherwise.

```
#include<stdio.h>
int ack(int,int);
void main( )
{int x,y;
clrscr();
printf("Enter 2 positive integers : ");
scanf("%d%d",&x,&y);
printf("ANSWER: %d ",ack(x,y) );
getch( );
}
int ack(int m,int n)
{ if (m==0)
    return n+1;
    else if(n==0 && m>0)
        return ack(m-1,1) ;
    else
        return ack(m-1,ack(m,n-1));
}
```


⊛ Entered variable must be declared once more before void main

F.E SEM-II

Structured Programming Approach

C5-P12:- FIND OUTPUT.

```
#include<stdio.h>
static int i=5;
void increment()
{ static int i = 1;
printf("i= %d ",i);
i++;
}
int main()
{ clrscr();
increment();
increment();
increment();
getch();return (0);
}
```

Output

i = 1 i = 2 i = 3

⇒ local variable given
pref. over global variable
int

C5-P13:-FIND OUTPUT

```
#include<stdio.h>
void f1()
{extern int n3;
static int n1;
int n2=20;
n1=n1+10;
n2=n1+n2;
n3=n1+n2;
printf("%d %d %d \n",n1,n2,n3);
}
int n3;
int main( )
{ register int i;
clrscr();
for(i=1;i<=3;i++)
f1();
getch();return (0);
}
```

Output:

10 30 40
20 40 60
30 50 80

for loop 1st run
n1 = 0 + 10 = 10
n2 = 10 + 20 = 30
n3 = 10 + 30 = 40
print 10 30 40

for loop 2nd run
n1 = 10 + 10 = 20
n2 = 20 + 20 = 40
n3 = 20 + 40 = 60
print 20 40 60

for loop 3rd run
n1 = 20 + 10 = 30
n2 = 30 + 20 = 50
n3 = 30 + 50 = 80
print 30 50 80

Pointers and Function

5.11 Call by address (call by reference)

The two-way communication between the functions is achieved using pointers.

The pointer variables occupy different memory locations.

```
void swap (int *p, int *q)
```

```
{
int temp;
temp = *p;
*p = *q;
*q = temp;
}
```

It is passing a pointer
as an argument to the fⁿ

While calling the function, the address of variables are passed as parameters.

The corresponding call will be given as, swap (&x,&y);

5.12 CALL BY VALUE MECHANISM

In call by value, a copy of the data is made and the copy is sent to the function. The copies of the value held by the arguments are passed by the function call. Since only copies of values held in the arguments are passed by the function call to the formal

parameters of the called function, the value in the arguments remains unchanged. In other words, as only copies of the values held in the argument are sent to the formal parameters, the function cannot directly modify the arguments passed

⑤ C5-P14:-WAP to swap two variables to demonstrate call by value and call by address.

```
#include<stdio.h>
void swap_v(int,int);
void swap_a(int*,int*);
int main()
{int a=10,b=20;
clrscr();
printf("a= %d \t b= %d \n",a,b);
swap_a(&a,&b);
printf("after swap with add.a= %d \t b=%d \n",*i,*j);
swap_v(a,b);
printf("After swap with value a=%d \t b= %d \n",i,j);
getch();return(0);
}

void swap_v(int i,int j)
{int t;
t=i;
i=j;
j=t; printf("Within swap by value for a = %d \t b = %d \n",i,j);
}
```

```
void swap_a(int*i,int*j)
{int t;
t=*i;
*i=*j;
*j=t;} printf("Within swap by address for a = %d \t b = %d \n",*i,*j);
```

5.13 POINTER AND FUNCTIONS:

We have discussed how to pass arguments by value and by reference. Passing value by reference is nothing but passing a pointer as an argument to function.

Pointer to a function is pointer variable storing the address of the starting instruction or statement of the function. We know that when program executes it has to be loaded in to memory. The function which is named block of statements also gets its location in the memory. The pointer variable will contain the address of the first statement of the function from where it begins.

C5-P15:-

//pointer to function with argument

```
#include <stdio.h>
void main ()
{ float (*a)( float, int ); /* Pointer to Function declaration */
float add( float a, int b ) ; /*function declaration*/
float result ;
clrscr () ;
a = add; /* Pointer to Function assignment */
result = (*a)(12.7, 23) ; /* Function call using pointer 'a' */
printf("Result = %f",result);
getch () ;
}

float add(float a, int b ) /*function defination*/
```



```
{a = a + b ;
return ( a ) ;
}
```

Explanation of program:-

See that the pointer to function returning a float value is declared in the program as,
float (*a) (float ,int);

The declaration pattern of pointers to function is made as,

<return type of function> (*<pointer name>) (parameter);

Here, 'float' means that it is a pointer to a function which can return 'float' type value.

Also, observed that how an assignment of a pointer to function has been made. It is,
a=add;

The general pattern of assignment of a pointer to function is,

<pointer name>=<function name>;

This assigns the address of function i.e. address of the first statement of the function to the pointer variable.

5.14 FUNCTION RETURNING A POINTER:-

A function can return a pointer type of value; that means an address value. The use of return value of pointer type is illustrate in the program below:

C5-P16:-

```
#include <stdio.h>
void main ()
{ double (*a) ( ); /* Pointer to Function declaration */
  double function ( ) ; /*function declaration*/
  double result ;
  clrscr ( ) ;
  a = function; /* Pointer to Function assignment */
  result = (*a) ( ) ; /* Function call using pointer 'a' */
  printf("Result = %f",result);
  getch ( ) ;
}
double function ( ) /*function definition*/
{double a,b;
  a=10.5;
  b=20.3;
  a = a + b ;
  return ( a ) ;
}
```

5.15 Function accepting & returning a pointer

C5-P17:-

```
#include <stdio.h>
void main ()
{ int *square ( int *a ) ; /*function which accept pointer & return pointer*/
  int i, *result ;
  clrscr ( ) ;
  printf("i = ") ;
  scanf("%d",&i);
  result = square ( &i ) ;
  printf("square ( i ) : %d ",*result) ;
  getch ( ) ;
}
```



```
int *square ( int *a )
{ *a = *a * *a ;
  return ( a );
}
```

The program declares an integer variable 'i' whose square has to be found. It then calls the function square(). The call to square() is made using an address of 'i'. the formal parameter 'a' which is pointer variable, is used to compute the square of 'i'. See that, before the function name 'int*' is written which specifies that the type of the return value of the function square() is pointer to int. The function square() returns the data value of 'a', which is nothing but the address of 'i' i.e. a pointer type of value. This address value returned by the function square() gets assigned to the pointer variable 'result' which now points to 'i'. the main () function then uses 'result' pointer to print the value of 'i' pointed by 'result'

Practice Programs

C5-P18:-Develop a C program to calculate logarithmic and exponent values by giving a call to a function 'log_X'10 and 'expo'.

C5-P19:-A positive integer is entered through the keyboard. Write function to obtain the prime factors of this number.

C5-P20:-Write a function to find GCD and LCM of two integer using Euclid's algorithm.

C5-P21:-Write a function to calculate factorial of a number. Using the function calculate $[i] \frac{n!}{(n-r)!}$ $[ii] \frac{n!}{r!(n-r)!}$

C5-P22:-WAP to find the GCD of three numbers recursively, {Hint: GCD is defined for two numbers. To find GCD of three numbers call GCD function within a GCD function.

C5-P23:-Write a recursive function that accepts one positive integer say x and it should calculate and return sum of all the odd numbers between 1 to x. Write a suitable main function that will read one positive integer say n and will print sum of all the odd numbers between 1 to n using the recursive function.

C5-P24:-Write a recursive function that accepts one positive integer in decimal and will print the corresponding binary equivalent. Write a suitable main function that will accept positive integer from the user and will print its binary equivalent with the help of recursive function.

C5-P25:- WAP to reverse a number by recursion.

C5-P26:-Write a general purpose function to convert given year into roman equivalent

Decimal	Roman	Decimal	Roman
1	i	100	c
5	v	500	d
10	x	1000	m
50	l		

C5-P27:-Write a recursive function that accepts one positive integer in decimal and will print the corresponding hexadecimal equivalent. Write a suitable main function that will accept positive integer from the user and will print its hexadecimal equivalent with the help of the recursive function.

C5-P28:- WAP to print Fibonacci numbers by recursion.

C5-P29:- WAP to calculate compound interest and amount by formula

$A = P(1 + \frac{R}{100})^n$. P = principal amt., R = Rate of Interest, n = number

of years. Use User define function to calculate power. Program should accept P, R, n and display interest for each year.

10/4
C5-P30:-Write a Menu driven program, to perform arithmetic operations on two integers. The menu should be repeated until the user selects STOP option. Program should have independent user defined functions for each case.

Prof. Manirekar

CH.6 ARRAY

An array is a collection of similar elements. These similar element could be all int's, or all floats, or all chars, etc. Usually, the array of character is called a 'string', whereas an array of int's or floats is called simply an array. All elements of any given array must be of the *same type*.

[a] An Array is a collection of similar element.

[b] The first element in the array is numbered 0, so the last element is 1 less than the size of the array.

[c] An array is also known as a subscripted variable.

[d] Before using an array its type and dimension must be declared.

[e] However big an array its elements are always stored in continuous memory locations.

6.1 DECLARATION OF ONE-DIMENSIONAL ARRAYS

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

`type variable-name[size];`

The *type* specifies the type of element that will be contained in the array, such as int, float, or char and the size indicated the maximum number of elements that can be stored inside the array. For example,

`float height [50];`

declares the height to be an array containing 50 real elements.

- Any reference to the arrays outside the declared limits would not necessarily cause an error. Rather, it might result in unpredictable program results.

- The size should be either a numeric constant or a symbolic constant.

INITIALIZATION OF ONE-DIMENSIONAL ARRAYS

After an array is declared, its elements must be initialized. Otherwise, they will contain "garbage". An array can be initialized at either of the following stages

- At compile time
- At run time

Compile Time Initialization

The values in the list are separated by commas. For example, the statement

`int number [3] = { 0, 0, 0 };`

`float total [5] = { 0.0, 15.75, -10};`

Will initialize the first three element to 0.0, 15.75, and -10.0 and the remaining two elements to zero. The size may be omitted. In such cases, the compiler allocated enough space for all initialized elements. For example, the statement

`int counter [] = {1,1,1,1};`

will declare the counter array to contain four elements with initial values 1.

Run Time Initialization

```

        for (i = 0; i < 100; i = i + 1)
        { if    i < 50
sum[i] = 0.0;    /* assignment statement */
                else

```



```

        sum[i] = 1.0;
    }

```

We can also use a read function such as scanf to initialize an array. For example, the statements

```

int x [3];
scanf ("%d%d%d", &x[0], &x[1], &x[2]);

```

will initialize array elements with the values entered through the keyboard.

C6-P1:-

```

#include<stdio.h>
int main()
{ int i,j,sum=0;
  int marks[5];
  float ave;
  clrscr();
  for (i=0; i<=4; i++)
  { printf("Enter marks : ");
    scanf("%d",&marks[i]);
  }
  for (j=0; j<5; j++)
    sum = sum+marks[j];
  ave = sum/5;
  printf("Average marks : %f",ave);
  getch();return(0);
}

```

Output:-

```

Enter marks : 10
Enter marks : 20
Enter marks : 30
Enter marks : 40
Enter marks : 45
Average marks : 29

```

6.2 Array Element in Memory:-

Consider the following array declaration:- `int arr [8]`

16 bytes get immediately reserved in memory, 2 bytes each for the 8 integers (under Windows/ Linux the array would occupy 32 bytes as each integer would occupy 4 bytes.) And since the array is not being initialized, all eight values present in it would be garbage values. This so happens because the storage class of this array is assumed to be auto. If the storage class is declared to be static then all the array elements would have a default initial value as zero.

Initializing 2- Dimensional Arrays:-

```

int stud [4] [2] = {
    {1234, 56},
    {1212, 33},
    {1434, 80},
    {1312, 78},
};

```

or even this would work.....

```

int stud [4] [2] =
{1234,56,1212,33,1434,80,1312,78};

```

of course with a corresponding loss in readability.

	col. no. 0	col. no. 1
row no. 0	1234	56
row no. 1	1212	33
row no. 2	1434	80
row no. 3	1312	78

It is important to remember that while initializing a 2- d array it is necessary to mention the second (column) dimension, whereas the first dimension (row) is optional.

Thus the declarations,

```

int arr [2] [3] = {12,34,23,45,56,45};

```


`int arr [] [3] = {12,34,23,45,56,45};`
are perfectly acceptable.

6.3 Memory Map of 2 Dimensional Array:-

s[0][0]	s[0][1]	s[1][0]	s[1][1]	s[2][0]	s[2][1]	s[3][0]	s[3][1]
1234	56	1212	33	1434	80	1312	78
65508	65510	65512	65514	65516	65518	65520	65522

6.4 3 Dimensional Array or Multi-Dimensional Array:-

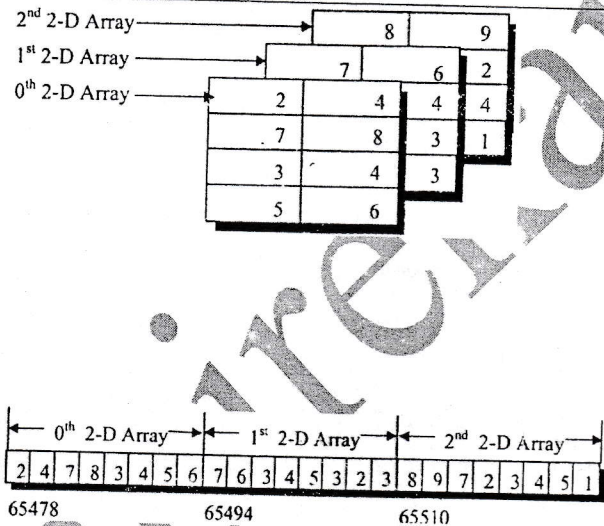
`int arr[3][4][2]= {`

```

{2, 4},
{7, 8},

        {3, 4},
        {5, 6}
    },
    {
        {7, 6},
        {3, 4},
        {5, 3},
        {2, 3}
    },
    {
        {8, 9},
        {7, 2},
        {3, 4},
        {5, 1}
    }
};

```



A Multi-Dimensional array can be thought of as an array of array of arrays. How would you refer to the array element 1 in the above array? The first subscript should be [2], since the element is in third two dimensional array; the second subscript should be [3] since the element is the fourth row of the two - dimensional array; and the third subscript should be [1] since the element is in second position in the one - dimensional array, We can therefore say that the element 1 can be referred as `arr [2] [3] [1]`.

6.5 Searching and Sorting

Sorting is the process of arranging elements in the list according to their values, in ascending or descending order. A sorted list is called an *ordered list*. Sorted lists are especially important in list searching because they facilitate rapid search operations. Many sorting techniques are available. The three simple and most important among them are:

- Bubble sort (sinking sort)
- Selection sort
- Insertion sort

Exchange Sort
Selection Sort
Bingo Sort

Other sorting techniques include Shell sort, Merge sort and Quick sort.

Searching is the process of finding the location of the specified element in a list. The specified element is often called the search key. If the process of searching finds a match of the search key with a list element value, the search said to be successful; otherwise, it is unsuccessful. The two most commonly used search techniques are:

- Sequential search (Linear Search) (non-sorted array works)
- Binary search (array needs to be sorted)

C6-P2:- Accept numbers from user & arrange them into an array. The number to be searched is entered through the keyboard by the user. Write a program to find if the number to be searched is present in the array and if it is present, display the number of times it appears in the array. Write a program to find out how many of them are positive, how many are negative, how many are even and how many odd.

C6-P3:- Write a program to copy the contents of one array into another in the reverse order.

C6-P4:- If an array arr contains n elements, then write a program to check if $arr[0] = arr[n-1]$, $arr[1] = arr[n-2]$ and so on.

Enter no. of element in array : 8

Enter elements in array : 1 2 3 4 4 3 2 1

Matching elements : 8

C6-P5:- WAP to print Fibonacci series

C6-P6:- WAP to find largest number and its position in array.

C6-P7:- WAP to find second largest number in array.

C6-P8:- WAP in c to cyclically rotate the elements in array. Program should accept a choice in which direction to rotate i.e. left or right. Depending on choice it should perform cyclic rotation. Suppose array a contains elements {1,2,3,4,5} then if choice is rotate right o/p should be {5,1,2,3,4} and if choice is rotate left then o/p should be {2,3,4,5,1}.

C6-P9:- Write a program that accepts one dimensional array of not more than 50 floating point numbers. The program should calculate and print their mean, variance and standard deviation. chp6

$$\text{Mean} = \frac{\sum_{i=1}^n X_i}{n} \quad \text{Variance} = \frac{\sum_{i=1}^n (X_i - \text{mean})^2}{n} \quad \text{Std. deviation} = \sqrt{\text{variance}}$$

C6-P10:- WAP to calculate following series

$$\sum_{i=1}^n x_i^2 - \left[\sum_{i=1}^n x_i \right]^2$$

C6-P11:- Implement the selection sort algorithms.

Selection Sort

Iteration 1

0	44
1	33
2	55
3	22
4	11

Iteration 2

0	11
1	44
2	55
3	33
4	22

Iteration 3

0	11
1	22
2	55
3	44
4	33

Iteration 4

0	11
1	22
2	33
3	55
4	44

Result

0	11
1	22
2	33
3	44
4	55

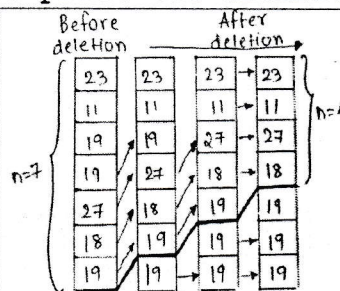
```
for(i=0; i<n-1; i++)
{
    min=i;
    for(j=i+1; j<n; j++)
    {
        if(arr[j]<arr[min])
        {
            min=j;
        }
    }
    temp=arr[i];
    arr[i]=arr[min];
    arr[min]=temp;
}
```

C6-P11 on pg 81
 $arr[i] = arr[min];$
 $arr[min] = temp;$

C6-P12:- WAP to delete duplicate elements from array.

C6-P13:- WAP that accepts array of not more than 100 integers. The program should also accept element x which is to be inserted into an array and it should also accept position j at which the element x is to be inserted. The program should print the modified array.

C6-P14:- Write a program that accepts array of not more than 100 integers. The program should also accept element x which is to be deleted from an array. Delete all the occurrence of the element x. The program should print the modified array.



C6-P15:- Transform 2D array to 1D array:

C6-P16:- Pascal: The following set of number is called 'Pascal's Triangle'.

1						
1	1					
1	2	1				
1	3	3	1			
1	4	6	4	1		
1	5	10	10	5	1	

④ you need to print 1 yourself
the formula prints the inner thing
→ if you don't print 1, garbage value will print

If we denote rows by i and columns by j, then any element in the triangle is given by- $P[i][j] = P[i-1][j-1] + P[i-1][j]$

C6-P17:- WAP that accepts a square matrix of order not more than 10 * 10. Your program should calculate and print sum of both diagonal elements, lower triangular and upper triangular elements independently.

C6-P18:- WAP that accepts a square matrix of order not more than 10 * 10. Your program should obtain its transpose into the same array and print it.

C6-P19:- WAP that accepts a square matrix of order not more than 10 * 10. Your program should determine and print whether the matrix is symmetrical or not. (without using second matrix) → $(n^2 - n) / 2$

C6-P20:- WAP for matrix addition and multiplication. (40%)

Syllabus topics: Pointers & Array, Passing Array to Function, Pointers and Two dimensional Array, Array of Pointers, Dynamic Memory Allocation.

6.6 POINTERS AND ARRAYS

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

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


Elements	→	x[0]	x[1]	x[2]	x[3]	x[4]
Value	→	1	2	3	4	5
Address	→	1000	1002	1004	1006	1008

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

Rules of pointer Operations

The following rules apply when performing operations on pointer variables.

1. A pointer variable can be assigned the address of another variable.
2. A pointer variable can be assigned the values of another pointer variable.
3. A pointer variable can be initialized with NULL or zero value.
4. A pointer variable can be pre- fixed or post- fixed with increment or decrement operators.
5. An integer value may be added or subtracted from a pointer variable.
6. When two pointers point to the objects of the same data types, they can be compared using relational operators.
7. A pointer variable cannot be multiplied by a constant.
8. Two pointer variables cannot be added.

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

$p = x;$

this is equivalent to - $p = \&x[0];$

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

The relationship between p and x is shown as:

$p = \&x[0] (=1000)$

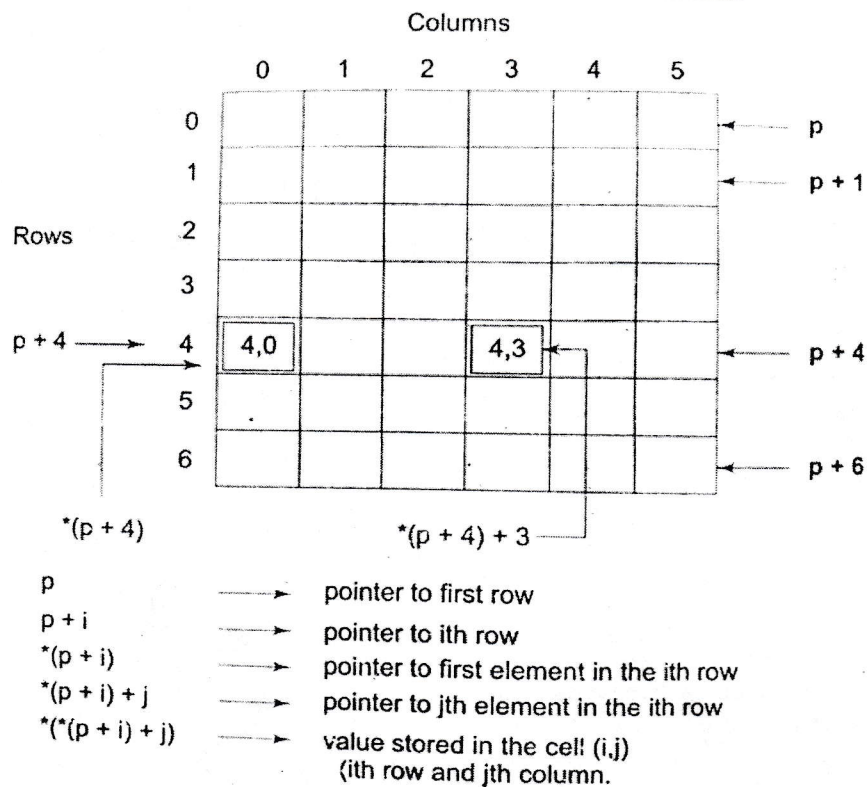
$p+1 = \&x[1] (=1002)$

$p+2 = \&x[2] (=1004)$

$p+3 = \&x[3] (=1006)$

$p+4 = \&x[4] (=1008)$

(i+p) or (p+i) is the same

6.7 Pointer for two dimensional array:

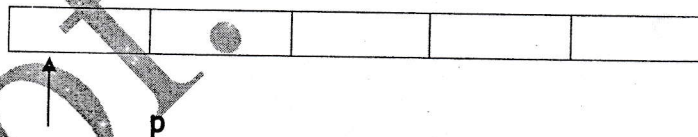
Q] What is p in the following declaration?

- 1] `float (*p)[5];`
- 2] `float*p[5];`

Ans:

1] `float (*p)[5];`

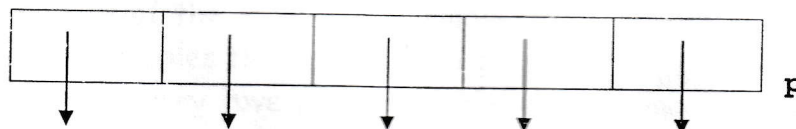
In this declaration, p is pointer to array of five floating point numbers. Consider the following diagram.



The elements of the above array can then be referred as $p[0]$, $p[1]$, $p[2]$, $p[3]$, and $p[4]$ and all these elements will be float values.

2] `float*p[5];`

In this declaration, p is array of 5 elements and all the elements are pointers to floating values. Consider the following diagram.



All the five elements $p[0]$, $p[1]$, $p[2]$, $p[3]$ and $p[4]$ are pointers to float.

C6-P21:- find output.

```
#include<stdio.h>
void main( )
{int*p,i[3];
clrscr();
i[0]=0;
i[1]=1;
i[2]=2;
p=&i[1];
printf("*p++ = %d ",*p++);
getch();
}
```

Output:-

~~*p++ = 2~~

*p++ = 1

(*p++ means print *p and then increment it)

C2-P22:- find output.

```
#include<stdio.h>
#include<conio.h>
void main()
{ int a[]={10,20,30,40,50};
  int *j,*k;
  j=&a[4];
  k=(a+4);
  if(j==k)
  printf("Both pointers points to
  same location");
  else
  printf("Both pointers does not
  points to same location");
  getch();
}
```

Output:-

~~Both~~
same loc

add of a[0]
k = a + 4 ⇒ *k = 50
j = 4a[4] ⇒ *j = 50

C2-P23:-

```
#include<stdio.h>
int main()
{ int i,a[2]={10,20};
clrscr();
for(i=0;i<=1;i++)
{printf("%d\n",a[i]);
printf("%d\n",*(a+i));
printf("%d\n",*(i+a));
}
getch();return(0);
}
```

10 → a[0] = 10
10 → *(a+0) = *a = a[0] = 10
10 → *(0+a) = *a = a[0] = 10
20 → a[1] = 20
20 → *(a+1) = a[1] = 20
20 → *(1+a) = a[1] = 20

C6-P24:- WAP to find sum of 10

float values by using pointer.

```
#include<stdio.h>
void main()
{ float *ptr;
  float array[10];
  float sum = 0;
  int i,j;
  ptr = array;
  clrscr();
  printf("Enter 10 float values : ");
  for(i = 0; i < 10 ; i++)
  { scanf("%f",&array[i]);
    sum = sum + *(ptr+i);
  }
}
```

Output:-

next 02


```

for(j=0; j<10 ;j++)
{
printf("%f \n", *(ptr+j));
}
printf("sum = %f", sum);
getch();
}

```

Enter 10 fms — — — — —

—
—
—
—

Sum = —

6.8 DYNAMIC MEMORY ALLOCATION

C language requires the number of elements in an array to be specified at compile time. The process of allocating memory at run time is known as dynamic memory allocation. Although C does not inherently have this facility, there are four library routines known as "memory management functions" that can be used for allocation and freeing memory during program execution.

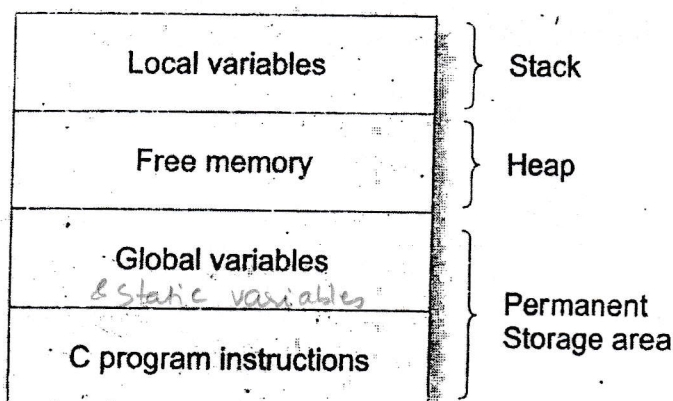
Memory Allocation Functions

Function	Task
malloc	Allocates request of bytes and returns a pointer to the first byte of the allocated space
calloc	Allocates space for an array of elements, initializes them to zero and then returns a pointer to the memory.
free	Frees previously allocated space
realloc	Modifies the size of previously allocated space

Memory Allocation Process

The conceptual view of storage of a C program in memory.

Storage of a C program



The program instructions and global and static variables are stored in a region known as *permanent storage area* and the local variables are stored in another area called *stack*. The memory space that is located between these two regions is available for dynamic allocation during execution of the program. This free memory region is called the *heap*. The size of the heap keeps changing when program is executed due to creation and death of variables that are local to functions and blocks. Therefore, it is possible to encounter memory "overflow" during dynamic allocation process. In such situations, the memory allocation functions mentioned above return a NULL pointer (when they fail to locate enough memory requested).

ALLOCATING A BLOCK OF MEMORY: MALLOC

A block of memory may be allocated using the function `malloc`. The `malloc` function reserves a block of memory of specified size and returns a pointer of type `void`. This means that we can assign it to any type of pointer. It takes the following form:

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

`ptr` is a pointer of type `cast-type`. The `malloc` returns a pointer (of `cast type`) to an area of memory with size `byte-size`.

Example:

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

On successful execution of this statement, a memory space equivalent to "100 times the size of an `int`" bytes is reserved and the address of the first byte of the memory allocated is assigned to the pointer `x` of type of `int`.

6.10 ALLOCATING MULTIPLE BLOCKS OF MEMORY: CALLOC

`calloc` is another memory allocation function that is normally used for requesting memory space at run time for storing derived data types such as arrays and structures. While `malloc` allocates a single block of storage space, `calloc` allocates multiple blocks of storage, each of the same size, and then sets all bytes to zero. The general form of `calloc` is:

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

The above statement allocated contiguous space for `n` blocks, each of `elem-size` bytes. All bytes are initialized to zero and a pointer to the first byte of the allocated region is returned. If there is not enough space, a `NULL` pointer is returned.

```
struct student
{
    char name[25];
    float age;
    long int id_num;
};
typedef struct student record;
record *st_ptr;
int class_size = 30;

str_ptr=(record*)calloc(class_size, sizeof(record));
```

`record` is of the struct `student` having three members: `name`, `age` and `id_num`. The `calloc` allocates memory to hold data for 30 such records. We must be sure that the requested memory has been allocated successfully before using the `st_ptr`. This may be done as follows:

```
if(st_ptr == NULL)
{
    printf("Available memory not sufficient");
    exit(1);
}
```


6.11 RELEASING THE USED SPACE: FREE

With the dynamic run-time allocation, it is our responsibility to release the space when it is not required. The release of storage space becomes important when the storage is limited.

When we no longer need the data we stored in a block of memory, and we do not intend to use that block for storing any other information, we may release that block of memory for further use, using the **free** function:

free (ptr);

ptr is a pointer to a memory block which has already been created by **malloc** or **calloc**.

6.12 ALTERING THE SIZE OF A BLOCK: REALLOC

We can change the memory size already allocated with the help of the function **realloc**. This process is called the *reallocation* of the memory. For example, if the original allocation is done by the statement

ptr = malloc(size);

then reallocation of space may be done by the statement.

ptr = realloc(ptr, newsize);

This function allocates a new memory space of size *newsize* to the pointer variable *ptr* and returns a pointer to the first byte of the new memory block. The *newsize* may be larger or smaller than the *size*.

Q-P25:- by using dynamic memory allocation write a program to read and store N integers in an array, where value of N is defined by user. Find minimum and maximum numbers from the array.

```
#include<stdio.h>
#include<conio.h>
void main()
{ int *ptr1,*ptr2;
  int i,j,e,temp;
  printf("Enter size of Array:\n");
  scanf("%d",&e);
  ptr1 = (int*)malloc(e*sizeof(int));
  ptr2=ptr1;
  for(i=0;i<e;i++)
  {printf("Enter no ");
  scanf("%d",ptr1);
  ptr1++;
  }
  for(i=0;i<e-1;i++)
  for(j=i+1;j<e;j++)
  if(*(ptr2+i)>*(ptr2+j))
  { temp=*(ptr2+i);
    *(ptr2+i)=*(ptr2+j);
    *(ptr2+j)=temp;
  }
  printf("Min:%d & Max:%d",*ptr2,*(ptr2+(e-1)));
  getch();
  free(ptr1);
}
```

by cast to ptr

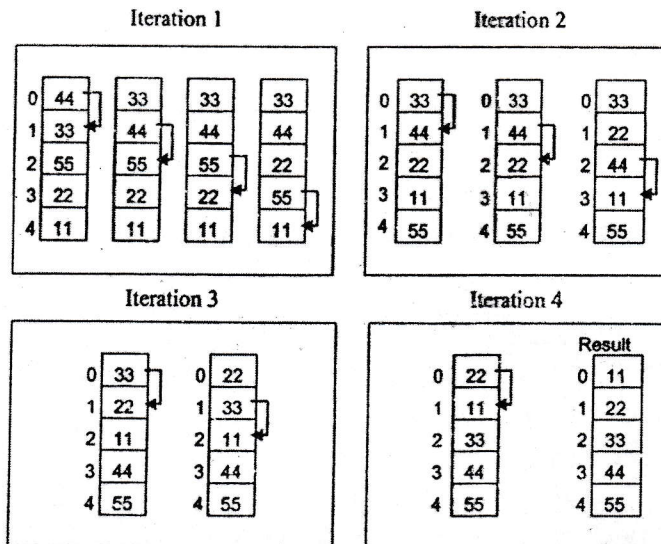
exchange selection sort

optional

Practice Programs

C6-P26:-Implement the bubble sort algorithms.

Bubble Sort



C6-P27:-Write a function that accepts the floating point array and the function should also accept the option. If the option is one then the array should be sorted in the ascending order and if the option is 2 then the array should be sorted in the descending order. Write suitable main function.

C6-P28:- WAP for binary search.

C6-P29:- WAP that accepts any positive long integer as amount in rupees. The program should calculate and print minimum number of notes required to meet this amount. Use Indian currency.

C6-P30:- WAP that accepts the date in dd/mm/yyyy format and the program should print day number of the year.

C6-P31:- WAP that accepts a square matrix of order not more than 10 * 10. Your program should determine and print whether the matrix is symmetrical or not. (By Using second matrix) → A^T

~~C6-P32:- WAP to multiply 2 matrices after checking compatibility. Your program should make use of function to accept element of matrix, display of matrix, and multiply matrix.~~

C6-P33:-Write a program that accepts a matrix of order not more than 10 * 10. Your program should print all the elements of the matrix. The program should print sum of all the elements in each row along with that row. It should also print the sum of all the elements in each column below that column. The program should also print the sum of all the elements below the row sums.

C6-P34:-

Four experiments are performed, each experiments consisting of six test results. The result for each experiment follows. Write a C program to compute and display the average the test results for each experiment.

1 st experiment results	23.2	31.5	16.9	28.0	26.3	28.2
2 nd experiment results	34.8	45.2	20.8	39.4	33.4	36.8
3 rd experiment results	19.4	50.6	45.1	20.8	50.6	13.4
4 th experiment results	36.9	42.7	20.8	10.2	16.8	42.7

C6-P35:-Write a c program that declares three single dimensional arrays named price, quantity and amount. Each array should be

declared in main () and should be capable of holding ten double precision numbers. The numbers that should be stored in price 10.20, 11.30, 13.14, 16.9, 18.1, 2.71, 7.55, 15.12, 9.45, 17.0. The numbers that should be stored in quantity are 3, 9.7m, 6.40, 4.5, 5.6, 6.2, 7, 2.8, 15.0, 18.0. Your program should pass these three array to a function named extend (), which should calculate the elements in the amount array as the product of elements in the price and quantity arrays. After extend () has put values into the array, the values in the array should be display from within main ().

C6-P36:- WAP to convert decimal number into binary number.

C6-P36:- WAP to convert decimal number into Octal number.

C6-P37:- WAP to convert decimal number into Hexadecimal number.

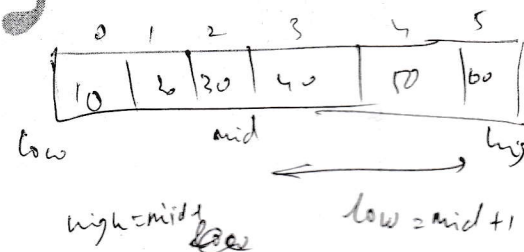
C6-P38:- WAP to convert binary to decimal.

C6-P39:- WAP to convert Octal to decimal.

C6-P40:- WAP to convert Hexadecimal to decimal.

C6-P41:- write a program that creates two integer arrays of size 8 and 7. Initialize the arrays with random values. Sort the arrays in ascending order with the help of a user defined function namely 'sortArray'. Merge these arrays with the help of another user defined function named 'merge Arrays' which returns a new array. Program should display the arrays before and after sorting. Also the merged array.

price[10] ✓
qty[10] ✓
amt[10] ← *mid array* → main
extend()
amt[i] = price[i] * qty[i];
~~amt[10] = price[10] * qty[10];~~
}



CH7. STRING

A string is a sequence of characters that is treated as a single data item. Character strings are often used to build meaningful and readable programs. The common operations performed on character strings include:

- Reading and writing strings
- Combining strings together
- Copying one string to another
- Comparing strings for equality
- Extracting a portion of a string

7.1 DECLARING AND INITIALIZING STRING VARIABLES

C permits a character array to be initialized in either of the following two forms: *null char*

```
char city [9] = "NEW YORK";
```

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

C also permits us to initialize a character array without specifying the number of elements. In such cases, the size of the array will be determined automatically, based on the number of elements initialized. For example, the statement

```
char string [ ] = {'G', 'O', 'O', 'D', '\0'};
```

We can also declare the size much larger than the string size in the initializer. That is, the statement.

```
char str[10] = "GOOD";
```

"why do we need a terminating null character? A string is not a data type in C, but it is considered a data structure stored in an array. The string is a variable-length structure and is stored in a fixed-length array. The array size is not always the size of the string and most often it is much larger than the string stored in it. Therefore, the last element of the array need not represent the end of the string. We need some way to determine the end of the string data and the null character serves as the "end-of-string" marker.

7.2 READING STRINGS FROM TERMINAL

Using **scanf** Function

The input function **scanf** can be used with **%s** format specification to read in a string of characters. Example:

```
char address[10];
scanf("%s", address);
```

The problem with the **scanf** function is that it terminates its input on the first white space it finds. (A white space includes blanks, tabs, carriage returns, form feeds, and new lines.) Therefore, if the following line of text is typed in at the terminal,

```
NEW YORK
```

Then only the string "NEW" will be read into the array **address**, since the blank space after the word 'NEW' will terminate the string reading. In the case of character arrays, the ampersand (&) is not required before the variable name.

We can also specify the field width using the form **%ws** in **scanf** statement for reading a specified number of characters from the input string. Example:

```
scanf("%ws", name);
```

Here, two things may happen.

1. The width **w** is equal to or greater than the number of characters type in. The entire string will be stored in the string variable.

2. The width w is less than the number of the characters in the string. The excess characters will be truncated and left unread.

7.3 Reading a Line of Text

A `scanf` function can input strings containing more than one character. Following are the specifications for reading character strings:

`%ws` or `%wc`

However, `%c` may be used to read a single character when the argument is a pointer to a char variable.

Some versions of `scanf` support the following conversion specifications for strings:

`%[characters]`

`%[^characters]`

The specification `%[characters]` means that only the characters specified within the brackets are permissible in the input string. If the input string contains any other character, the string will be terminated at the first encounter of such a character. The specification `%[^characters]` does exactly the reverse. That is, the characters specified after the circumflex (^) are not permitted in the input string. The reading of the string will be terminated at the encounter of one of these characters.

```
char line [80];
```

```
scanf ("%[^\\n]", line);
```

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

will read a line of input from the keyboard and display the same on the screen.

7.4 `gets ()` and `puts ()` : The usage of function `gets ()` and `puts ()` is shown below:

C7-P1:-

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

```
int main( )
```

```
{
```

```
char str[20];
```

```
clrscr();
```

```
printf("Enter any string :") ;
```

```
gets(str);
```

```
puts( " The string is : " ) ;
```

```
puts(str) ;
```

```
getch( ) ; return(0);
```

```
}
```

Output:

Enter any string :

Hello world

The string is :

Hello world

In this program `puts ()` can display only one string at a time. Also, on displaying a string, unlike `puts ()` places the cursor on the next line. Through `gets ()` is capable of receiving only one string at a time, it can receive a multi-word string.

7.5 STANDERD LIBRARY STRING FUNCTION:

STRING HANDLING FUNCTION	WHAT IT DOES?
<code>strlen (S1)</code>	It finds length of string, S1 excluding null character and returns an integer value.
<code>strlwr (S1)</code>	It converts the strings, S1 to a lower case string, S1.
<code>strcat (S1,S2)</code>	It appends or attached the string S2, at the end of the string S1 and places a null character at the end of this modified string, S1.

CH7. STRING

A string is a sequence of characters that is treated as a single data item. Character strings are often used to build meaningful and readable programs. The common operations performed on character strings include:

- Reading and writing strings
- Combining strings together
- Copying one string to another
- Comparing strings for equality
- Extracting a portion of a string

7.1 DECLARING AND INITIALIZING STRING VARIABLES

C permits a character array to be initialized in either of the following two forms: *null char*

```
char city [9] = "NEW YORK";
```

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

C also permits us to initialize a character array without specifying the number of elements. In such cases, the size of the array will be determined automatically, based on the number of elements initialized. For example, the statement

```
char string [ ] = {'G', 'O', 'O', 'D', '\0'};
```

We can also declare the size much larger than the string size in the initializer. That is, the statement.

```
char str[10] = "GOOD";
```

"why do we need a terminating null character? A string is not a data type in C, but it is considered a data structure stored in an array. The string is a variable-length structure and is stored in a fixed-length array. The array size is not always the size of the string and most often it is much larger than the string stored in it. Therefore, the last element of the array need not represent the end of the string. We need some way to determine the end of the string data and the null character serves as the "end-of-string" marker.

7.2 READING STRINGS FROM TERMINAL

Using **scanf** Function

The input function **scanf** can be used with %s format specification to read in a string of characters. Example:

```
char address[10]
scanf("%s", address);
```

The problem with the **scanf** function is that it terminates its input on the first white space it finds. (A white space includes blanks, tabs, carriage returns, form feeds, and new lines.) Therefore, if the following line of text is typed in at the terminal,

NEW YORK

Then only the string "NEW" will be read into the array **address**, since the blank space after the word 'NEW' will terminate the string reading. In the case of character arrays, the ampersand (&) is not required before the variable name.

We can also specify the field width using the form %ws in **scanf** statement for reading a specified number of characters from the input string. Example:

```
scanf("%ws", name);
```

Here, two things may happen.

1. The width **w** is equal to or greater than the number of characters type in. The entire string will be stored in the string variable.

2. The width w is less than the number of the characters in the string. The excess characters will be truncated and left unread.

7.3 Reading a Line of Text

A `scanf` function can input strings containing more than one character. Following are the specifications for reading character strings:

`%ws` or `%wc`

However, `%c` may be used to read a single character when the argument is a pointer to a char variable.

Some versions of `scanf` support the following conversion specifications for strings:

`%[characters]`

`%[^characters]`

The specification `%[characters]` means that only the characters specified within the brackets are permissible in the input string. If the input string contains any other character, the string will be terminated at the first encounter of such a character. The specification `%[^characters]` does exactly the reverse. That is, the characters specified after the circumflex (^) are not permitted in the input string. The reading of the string will be terminated at the encounter of one of these characters.

```
char line [80];
```

```
scanf ("%^[^n]", line);
```

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

will read a line of input from the keyboard and display the same on the screen.

7.4 `gets ()` and `puts ()` : The usage of function `gets ()` and `puts ()` is shown below:

C7-P1:-

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

```
int main ( )
```

```
{
```

```
char str[20];
```

```
clrscr();
```

```
printf("Enter any string :") ;
```

```
gets(str);
```

```
puts( " The string is : " ) ;
```

```
puts(str) ;
```

```
getch ( ) ;return(0);
```

```
}
```

Output:

Enter any string :

Hello world

The string is :

Hello world

In this program `puts ()` can display only one string at a time. Also, on displaying a string, unlike `puts ()` places the cursor on the next line. Through `gets ()` is capable of receiving only one string at a time, it can receive a multi-word string.

7.5 STANDARD LIBRARY STRING FUNCTION:

STRING HANDLING FUNCTION	WHAT IT DOES?
<code>strlen (S1)</code>	It finds length of string, S1 excluding null character and returns an integer value.
<code>strlwr (S1)</code>	It converts the strings, S1 to a lower case string, S1.
<code>strcat (S1,S2)</code>	It appends or attached the string S2, at the end of the string S1 and places a null character at the end of this modified string, S1.

S1-S2

<code>strcmp (S1, S2)</code>	It compares the two strings S1 and S2. It returns as integer value n as, $n < 0$, if $S1 < S2$ $n = 0$, if $S1 = S2$ $n > 0$, if $S1 > S2$
<code>strcpy (S1, S2)</code>	It copies the string S2 in to string S1, <u>modifying the string S1</u> .
<code>stricmp (S1, S2)</code>	It compares the two strings, S1 and S2 without regards to the case. The "i" indicates that this function ignores case.
<code>stricmp (S1, S2)</code>	It is identical to the function <code>stricmp (S1, S2)</code> . These both functions, return same result as for <code>strcmp (S1, S2)</code> .
<code>strupr (S1)</code>	It converts the string S1 to the uppercase string, S1.
<code>strchr (S1, c)</code> ↑	It searches for the <u>first</u> occurrence of character, c in the string, S1. If found it returns a pointer to the character, else returns a null pointer.
<code>strrchr (S1, c)</code> ↑	It searches for the <u>last</u> occurrence of the character, c in the string, S1. If found it returns a pointer to the character, else returns to the null pointer.
<code>strstr (S1, S2)</code>	It searches for the first occurrence of sub-string S2, in the other string, S1. On success, it returns a pointer to the element in S1 where S2 be begins.
<code>strset (S1, c)</code>	It sets all the characters in the string, S1 to the character, c. It quits when the terminating null character is detected and returns a modified string, S1.
<code>strncat (S1, S2, n)</code>	It appends utmost n characters of string S2 to the end of string S1 and then appends a terminating null character. It returns, the modified string, S1. The maximum length of modified string S1 is, $strlen(S1) + n$.
<code>strncmp (S1, S2, n)</code>	It compares the first n character of two strings, S1 and S2 and returns value similar to <code>strcmp (S1, S2)</code> .
<code>strncmpi (S1, S2, n)</code>	It functions similar to <code>strncmp (S1, S2, n)</code> , but without case sensitivity.
<code>strncpy (S1, S2, n)</code>	It copies utmost n character of string S2, into string, S1. It returns the modified string S1.
<code>strnset (S1, c, n)</code>	It copies the character c, into the first n places of the string, S1. If $n > strlen(S1)$, $strlen(S1)$ replaces n. Copying terminates when n characters have been set or when a null character is detected. It returns the modified string, S1.
<code>strrev (S1)</code>	It reverses all characters in the string, S1 except null character. It returns reversed string, S1.

7.6 PASSING STRINGS TO FUNCTIONS

Because the strings are treated as character arrays in C, the rules for passing strings to functions are very similar to those for passing arrays to functions.

1. The string to be passed must be declared as a formal argument of the function when it is defined. Example:

```
void display(char item_name)
```



```
{
    . . . . .
}
```

2. The function prototype must show that the argument is a string. For the above function definition, the prototype can be written as

```
void display(char str[ ]);
```

3. A call to the function must have a string array name without subscripts as its actual argument.

Example: `display (names);`

where `names` is a properly declared string array in the calling function.

7.7 POINTERS AND CHARACTER STRING

```
char str[5] = "good" ;
```

The compiler automatically insert the null character '`\0`' at the end of the string. C supports an alternative method to creates string using pointer variables of type `char`.

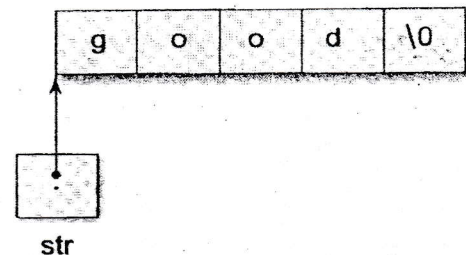
Example

```
char *str = "good" ;
```

This creates a string for the literal and then stores it address in the pointer variable `str`.

The pointer `str` now points to the first character of the string "good".

Note that `str` is a pointer, not a string.



7.8 ARRAY OF POINTERS

```
Char *name[3] = {
    "New Zealand" ,
    "Australia" ,
    "India"
};
```

Declares `name` to be an array of three pointers to characters, each pointer pointing to a particular name as :

```
name [0] —→ New Zealand
name [1] —→ Australia
name [2] —→ India
```

This declaration allocates only 28 bytes, sufficient to hold all the characters as shown

N	e	w		Z	e	a	i	a	n	d	\0
A	u	s	t	r	a	i	i	a		\0	
I	n	d	i	a	\0						

7.9 Pass by Value versus Pass by Pointers

The technique used to pass data from one function to another is known as *parameter passing*. Parameter passing can be done in two ways.

- Pass by value (also known as call by value)

- Pass by Pointers (also known as call by pointers)

In pass by value, values of actual parameters are copied to the variables in the parameter list of the called function. The called function works on the copy and not on the original values of the actual parameters. This ensures that the original data in the called function cannot be changed accidentally.

In pass by pointers (also known as pass by address), the memory addresses of the variables rather than the copies of values are sent to the called function. In this case, the called function directly works on the data in the calling function and the changed values are available in the calling function for its use.

Pass by pointers method is often used when manipulating arrays and strings. This method is also used when we require multiple values to be returned by the called function.

C7-P2:-

```
#include<stdio.h>
#include<string.h>
int main()
{ char name1[25],name2[25],
  str3[]="THANE",word1[10],word2[10];
  int len;
  clrscr();
  printf("Enter name : ");
  gets(name1);
  puts(name1);

  len = strlen(name1);
  printf("\nLength of string = %d",len);

  strcpy(name2,name1);
  printf("\n name2 : %s ",name2);

  if ( strcmp (name1,name2) == 0 )
    printf("\nName1 & Name2 are same");

  strcat(name1,str3);
  printf("\n %s",name1);

  printf("\nEnter a word :");
  gets(word1);
  strcpy(word2,word1);
  strrev(word2);
  if ( strcmp (word1,word2) == 0)
    printf("\nYour word is a palindrome");
  else
    printf("Your word is not a palindrome");
  getch();return(0);}
```

Output:-

```
Enter name : vinayak
vinayak
Length of string = 7
name2 :vinayak
Name1 & Name2 are same
vinayakTHANE
Enter a word: madam
```


Your word is a palindrome

C7-P3:- Write a program that accepts the string and put it into the character array. The program should modify the string in such a way that first letter of word will be printed in capital letters. *→ NOT string*

C7-P4:- WAP that reads a number from the user where the user enters a comma in the input. Then print number without a comma.

C7-P5:- WAP that reads a word and print whether it is palindrome or not.

C7-P6:- WAP to remove extra blank spaces.

Input:- Hello__world Output:- Hello_world

C7-P7:- WAP that accepts a string made up of words and your program should reverse the order of words. *I/p: Hello World O/p: World Hello*

C7-P8:- WAP to calculate length of string without using library function.

C7-P9:- WAP to copy one string to another without using library function

C7-P10:- WAP to concatenate one string to another without using library function.

C7-P11:- WAP to arrange strings in alphabetical order.

Practice Programs

C7-P12:- Write a C function called delchar () that can be used to delete characters from a string. The function should take three arguments. The string name, the number of characters to delete and the starting position in the string where characters should be deleted.

C7-P13:- WAP to accept a multi-line text and count number of words, number of lines and number of characters in it.

C7-P14:- This program is to illustrate how user authentication is made before allowing the user to access the secured resources. It asks for the user name and then the password. The password that you enter will not be displayed, instead that character is replaced by '*'. *3 3*
012345

CH8. STRUCTURES AND UNIONS

C supports a constructed data type known as structures, a mechanism for packing data of different types. A structure is convenient tool for handling a group of logically related data items. The concept of a structure is analogous to that of a 'record' in many other languages. More examples of such structures are:

Time	:	seconds, minutes, hours
Date	:	day, month, year
Book	:	author, title, price, year
City	:	name, country, population
Address	:	name, door-number, street, city
Inventory	:	item, stock, value

Structures helps to organize complex data in a more meaningful way.

8.1 DEFINING A STRUCTURE

We can define a structure to hold this information as follows :

```
struct book_bank
{
    char    title[20];
    char    author[15];
    int     pages;
    float   price;
};
```

The keyword struct declares a structure to hold the details of four data fields, namely title, author, pages, and price. These fields are called structure elements or members. Each member may belong to a different type of data. book_bank is the name of the structure and is called as the structure tag. The tag name may be used subsequently to declare variable that have the tag's structure. Note that the above definition has not declare any variables. It simply describes a format called template to represent information.

1. The template is terminated with a semicolon.
2. While the entire definition is considered as a statement, each member is declared independently for its name and type in a separate statement inside the template.
3. The tag name such as book_bank can be used to declare structure variables of its type, later in the program.

Arrays Vs Structures

Both the arrays and structures are classified as structured data types as they provide a mechanism that enable us to access and manipulate data in a relatively easy manner. But they differ in a number of ways:

1. An array is a collection of related data elements of same type. Structure can have elements of different types.
2. An array is derived data type whereas a Structure is a programmer-defined one.
3. Any array behaves like a built-in data type. All we have to do is to declare an Array variable and use it. But in the case of a structure, first we have to design and declare a data structure before the variables of that type are declared and used.

8.2 DECLARING STRUCTURE VARIABLES

After defining a structure format we can declare variables of that type. A structure variable declaration is similar to the declaration of variable of any other data types.

```
struct book_bank
{
    char    title[20];
    char    author[15];
    int     pages;
    float   price;
};
```

Struct book_bank book1, book2, book3;

The declaration

```
struct book_bank
{
    char    title[20];
    char    author[15];
    int     pages;
    float   price;
} book1, book2, book3;
```

is valid. The use of tag name is optional here. For example:

```
Struct
{
    .....
    .....
    .....
} book1, book2, book3;
```

declare book1, book2, and book3 as structure variables representing three books, but does not include a tag name.

8.3 ACCESSING STRUCTURE MEMBERS

We can access and assign values to the member of structure in a number of ways. the members themselves are not variables. They should be linked to the structure variables in order to make them meaningful members. For example, the word title, has no meaning where as the phrase 'title of book3' has a meaning. The link between a member and a variable is established using the member operator '.' which is also known as 'dot operator' or 'period operator'. For example, book1.price is the variable representing the price of book1 and can be treated like any other ordinary variable.

```
strcpy(book1.title, "BASIC C") ;
strcpy (book1.author, " Prof. Manjrekar") ;
book1. page = 250 ;
book1. price = 120.50;
```

8.4 STRUCTURE INITIALIZATION

Like any other data type, a structure variable can be initialized at compile time.

```
main ( )
{struct st_record
    {int weight ;
      float height ;
    } ;
```



```

struct st_record student1 = {60, 180.75} ;
struct st_record student2 = {53, 170.60} ;
....
....
}

```

Another method is to initialize a structure variable outside the function as shown below :

```

struct st_record
{
    int weight ;
    float height ;
}
student1 = {60, 180.75} ;
main ( )
{
    Struct st_record student2 = {53, 170.60} ;
    ....
    ....
}

```

Note that the compile-time initialization of a structure variable must have the following elements :

1. The keyword struct
2. The structure tag name.
3. The name of variable to be declared.
4. The assignment operator = .
5. A set of value for the members of the structure variable, separated by commas and enclosed in braces.
6. A terminating semicolon.

8.5 COPYING AND COMPARING STRUCTURE VARIABLES

Two variables of the same structure type can be copied the same way as ordinary variables. If person1 and person2 belongs to the same structure, then the following statements are valid :

```

person1 = person2 ;
person2 = person1 ;

```

However, the statements such as

```

Person1 == person2
person1 != person2

```

are not permitted. C does not permit any logical operations on structure variables. In case, we need to compare them, we may do so by comparing members individually.

8.6 OPERATIONS ON INDIVIDUAL MEMBERS

As pointed out earlier, the individual members are identified using the member operator, the dot. A member with the dot operator along with its structure variable can be treated like any other variable name and therefore can be manipulated using expressions and operators.

```

if (student1.number == 111)
    student1.marks += 10.00;
float sum = student1.marks + student2.marks ;
student2.marks *= 0.5 ;

```


The following statements are valid :

```
student1.number+ +;
++student1.number;
```

8.7 ARRAYS OF STRUCTURES

```
struct marks
{
    int  subject1 ;
    int  subject2 ;
    int  subject3 ;
};
int main ( )
{
    struct marks student [3] = { {45, 68, 81},
    {75, 53, 69}, {57, 36, 71} };
}
```

This declares the student as an array of three elements student[0], student[1], and student[2]

An array of structures is stored inside the memory in the same way as a multi-dimensional array. The array student actually looks as shown in fig.

student [0].subject 1	45
.subject 2	68
.subject 3	81
student [1].subject 1	75
.subject 2	53
.subject 3	69
student [2].subject 1	57
.subject 2	36
.subject 3	71

The array student inside memory

8.8 ARRAYS WITHIN STRUCTURES

C permits the use of arrays as structure members. We can use single or multi dimensional arrays of type int or float.

```
struct marks
{int number ;
    float subject [3] ;
}student [2] ;
```

Here, the member subject contains three elements, subject[0], subject[1] and subject[2]. These elements can be accessed using appropriate subscripts. For example, the name

```
student[1].subject[2] ;
```

would refer to the marks obtained in the third subject by the second student.

8.9 STRUCTURES WITHIN STRUCTURES

Structures within a structure means nesting of structures. Nesting of structures is permitted in C.

```
struct salary
{char name ;
    char department ;
    struct
    {int dearness ;
        int house_rent ;
        int city;
    }allowance ;
}employee ;
```

The salary structure contains a member named allowance which itself is a structure with three members. The members contained in the inner structure namely dearness, house_rent, and city can be referred to as :

```
employee.allowance.dearness
```


employee.allowance.house_rent
employee.allowance.city

8.10 STRUCTURES AND FUNCTIONS

C supports the passing of structure values as arguments of functions. There are three method by which the values of a structure can be transferred from one function to another.

1. The first method is to pass each member of the structure as an actual argument of the function call. The actual arguments are then treated independently like ordinary variables.
2. The second method involves passing of a copy of the entire structure to the called function. Since the function is working on a copy of the structure, any changes to structure members within the function are not reflected in the original structure (in the calling function).
3. The third approach employs the concept called pointers to pass the structure as an argument. In this case, the address location of the structure is passed to the called function. The function can access indirectly the entire structure and work on it. This is similar to the way arrays are passed to function.
4. The general format of sending a copy of a structure to the called function is:

The called function takes the following form:

The called function must be declared for its type, appropriate to the data type it is expected to return.

```
data_type function_name(struct_type st_name)
{
    .....
    .....
    return(expression);
}
```

C8-P1:-WAP that will store information of players using structure. Sort & Display names of these players alphabetically and with respect to number of matches played. Accept name of a player and display his information. Also print list of all players who have played more than 50 matches.

C8-P2:-Design a structure 'st' to contain name, date of birth and total marks obtained. Define the structure "dob" to represent date of birth. WAP to read data for n students in a class and sort them in descending order of total marks.

```
#include<stdio.h>
#include<string.h>
struct dob
{int dd,mm,yy;
};
struct st
{char name [25];
  struct dob birthdate;
  int total;
} st[25],temp;
void main()
{int i,j,n;
printf("How many student : ");
scanf("%d",&n);
for(i=0;i<n;i++)
```

no variable for this struct declared here

variable d) 1st struct declared here and is a field of 2nd struct


```

{ printf("Enter name of student:");
scanf("%s",&st[i].name);
printf("birthdate: \n");
scanf("%d %d %d",&st[i].birthdate.dd,
&st[i].birthdate.mm,&st[i].birthdate.yy);
printf("Total marks : \n");
scanf("%d",&st[i].total);
}

for (i=0;i<n-1;i++)
    for (j=i+1;j<n;j++)
        { if (st[i].total<st[j].total)
            { temp=st[j];
st[j]=st[i];
st[i]=temp;
}
        }

printf("Student result : \n");
for (i=0;i<n;i++)
{printf("Name:%s\t birthdate:%d%d%d \tTotal:%d \n",
st[i].name,st[i].birthdate.dd,
st[i].birthdate.mm,st[i].birthdate.yy,st[i].total);
}

for (i=0;i<n-1;i++)
{ for (j=i+1;j<n;j++)
    { if (strcmp(st[i].name,st[j].name)>0 )
        { temp=st[j];
st[j]=st[i];
st[i]=temp;
}
    }
}

printf("Students alphabetical list : \n");
for (i=0;i<n;i++)
{printf("Name:%s\t birthdate:%d%d%d\t Total:%d \n",
st[i].name,st[i].birthdate.dd,
st[i].birthdate.mm,st[i].birthdate.yy,st[i].total);
}

getch();
}

```

Input

marks-wise
sorting
(exchange
selection
sort)alphabetic
sorting
(exchange
selection
sort)

C8-P3:- Design a structure time to contain hours & minutes. WAP to Add 2 time instance by creating variables of structure time.

```

#include<stdio.h>
#include<string.h>
struct time
{int  hour,min;
} T1,T2,total;

void addition(struct time t1,struct time t2);
void main()
{printf("Enter first time in hour & min : ");
scanf("%d %d",&T1.hour,&T1.min);
printf("Enter second time in hour & min : ");
scanf("%d %d",&T2.hour,&T2.min);
addition(T1,T2);
getch();
}

```

1 feet = 12 inches


```

void addition(struct time T1, struct time T2)
{
    int temp1, temp2;
    temp1 = (T1.min + T2.min) / 60;
    temp2 = (T1.min + T2.min) % 60;
    total.hour = T1.hour + T2.hour + temp1;
    total.min = temp2;
    printf("total time = %d hours & %d min",
           total.hour, total.min);
}

```

8.11 UNIONS

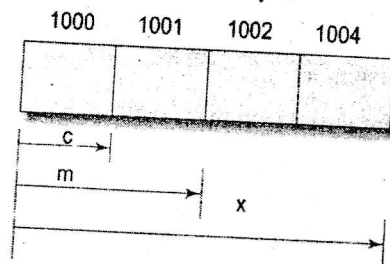
Unions are a concept borrowed from structures and therefore follows the same syntax as structures. However, there is major distinction between them in terms of storage. In structures, each member has its own storage location, whereas all the members of a union use the same location. This implies that, although a union may contain many members of different types, it can handle only one member at a time.

```

union item
{
    int m ;
    float x ;
    char c ;
} code ;

```

Storage of 4 bytes



Sharing of a storage location by union members

This declares a variable code of type union item. The union contains three members, each with a different data type. However, we can use only one of them at a time. This is due to the fact that only one location is allocated for a union variable, irrespective of its size.

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

To access a union member, we can use the same syntax that we used for structure members. That is,

```

code.m
code.x
code.c

```

in effect, a union creates a storage location that can be used by any one of its members at a time. Union may be used in all places where a structure is allowed. The notation for accessing a union member which is nested inside the structure remains the same as for the nested structures. Union may be initialized when the variable is declared. But, unlike structures, it can be initialized only with a value of the same type as the first member.

```

union item abc = {100} ;

```

is valid, but the declaration

```

union item abc = {10.75} ;

```

is invalid. This is because the type of the first member is int.

8.12 SIZE OF STRUCTURES

We normally use structures, unions and arrays to create variables of large sizes. The actual size of these variables in terms of bytes may change from machine to machine. We may use the unary operator sizeof to tell us the size of a structure (or any variable).

The expression

`sizeof (struct x)`

will evaluate the number of bytes required to hold all the members of the structure x.

C8-P4:- Wap to store the information of a person as his name or ID number using union. Ask the user for the information choice.

```
#include<stdio.h>
union info
{ char name[20];
  int ID;
};
void main()
{ union info I1;
  int choice;
  clrscr();
  printf("Enter choice:\n 1>Name \n 2>ID no.\n");
  scanf("%d",&choice);
  switch(choice)
  {case 1:printf("Enter your name:");
    scanf("%s",I1.name);
    break;
    case 2:printf("Enter your ID no.");
    scanf("%d",&I1.ID);
    break;
  default:printf("wrong choice");
  }
  if(choice==1)
  printf("Your name :%s",I1.name);
  else if(choice==2)
  printf("Your ID no :%d",I1.ID);
  getch();
}
```


Ch 9. File Management in C

Real-life problems involve large volumes of data and in such situations, the console oriented I/O operations pose two major problems.

1. It becomes cumbersome and time consuming to handle large volumes of data through terminals.
2. The entire data is lost when either the program is terminated or the computer is turned off.

It is therefore necessary to have a more flexible approach where data can be stored on the disks and read whenever necessary, without destroying the data. A file is placed on the disk where a group of related data is stored.

9.1 DEFINING AND OPENING FILE

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

Following is the general format for declaring and opening file:

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

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

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

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

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

1. When the mode is 'writing', a file with the specified name is created if the file does not exist. The contents are deleted, if the file already exists.
2. When the purpose is 'appending', the file is opened with the current contents safe. A file with the specified name is created if the file does not exist.
3. If the purpose is 'reading', and if it exists, then the file is opened with current contents safe otherwise an error occurs.

```
FILE *p1, *p2;  
p1 = fopen ("data", "r");  
p2 = fopen ("results", "w");
```

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

9.2 CLOSING A FILE

```
fclose (file_pointer);
```

This would close the file associated with the FILE pointer `file_pointer`.

```
FILE *p1, *p2;
p1 = fopen ("INPUT", "w");
p2 = fopen ("OUTPUT", "r");
. . . . .
. . . . .
fclose (p1);
fclose (p2);
```

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

9.3 INPUT/OUTPUT OPERATIONS ON FILES

The `getc` and `putc` Functions

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

writes the character contained in the character variable `c` to the file associated with FILE pointer `fp1`.

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

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

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

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

The `getw` and `putw` Functions

The `getw` and `putw` are integer-oriented functions. They are similar to `getc` and `putc` functions and are used to read and write integer values.

```
putw(integer, fp);
getw(fp);
```

The `fprintf` and `fscanf` Functions

```
fprintf (fp, "control string", list);
```

Where `fp` is a file pointer associated with a file that has been opened for writing. The *control string* contains output specifications for the items in the list. The *list* may include variables, constants and strings. Example:

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

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

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

```
fscanf (f2, "%s %d", item, &quantity);
```

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

C9-P1:- WAP to accept a set of characters from user until the user presses full stop('.') and store it in a text file. Read from the file and display the contents of the file.

```
# include<stdio.h>
void main()
{FILE *fp;
 char c=' ';
 clrscr();
 fp=fopen("test.txt","w");
 printf("Write data in the source file and press the full stop
 (.): \n");
 while(1)
 {c=getche();
 if(c=='.')
 {fputc(c,fp);
 break;
 }
 else
 fputc(c,fp);
 }
 fclose(fp);
 printf("\nYour source file:\n");
 fp=fopen("test.txt","r");
 while(!feof(fp))
 {printf("%c",getc(fp));
 }
 fclose(fp);
 getch();
 }
```

Handwritten notes:
 * while (1) ← infinite loop (always true)
 * while (0) ← loop will never run (always false)

C9-P2:- WAP to count the number of characters in a text file.

```
# include<stdio.h>
void main()
{ FILE *fp,*fp1;
 int n=0;
 char c=' ';
 clrscr();
 printf("Source file is:\n");
 fp=fopen("test.txt","r");
 while(!feof(fp))
 {printf("%c",getc(fp));
 }
 fclose(fp);

 fp1=fopen("test.txt","r");
 while(c=fgetc(fp1)!=EOF)
 {n++;
 }
 printf("\nNumber of char.in file : %d \n",n);
 fclose(fp1);
 getch();
 }
```

C9-P3:- WAP to accept the name and roll number of a students and store it in a text file. Read the stored data and display the same from the file. It should be menu driven program that can have multiple entries. The previous data should be retained and new data can be appended in the file. All the entries can be displayed if required.


```
# include<stdio.h>
void main()
{FILE *fp;
  char name[20];
  int roll,choice=0;
  clrscr();
  while(choice!=3)
  { printf("1.New Entry\n2.Display all entries\n3.Exit\nEnter your
choice :");
  scanf("%d",&choice);
  switch(choice)
  { case 1:
  fp=fopen("Student.txt","a");
  printf("Enter name and roll number of the student:");
  scanf("%s %d",name,&roll);
  fprintf(fp,"%s %d",name,roll);
  fclose(fp);
  break;
  case 2:
  fp=fopen("Student.txt","r");
  printf("Name\tRoll no.\n");
  while(!feof(fp))
  {
  fscanf(fp,"%s %d",name,&roll);
  printf("%s\t%d\n",name,roll);
  }
  fclose(fp);
  break;
  case 3: break;
  default: printf("Invalid Choice\n");
  }
  } //while loop ends here
  getch();
}
```

C9-P4:- WAP to copy text from one file to other after converting lower case to upper case and vice versa. Keep other characters as it is.

```
# include<stdio.h>
# include<conio.h>
void main() #include<ctype.h>
{FILE *source,*target;
  char sfile[20],tfile[20];
  char c,d;
  clrscr();
  printf("Enter name of source file\n");
  gets(sfile);
  printf("Enter name of target file\n");
  gets(tfile);

  source=fopen(sfile,"r");
  target=fopen(tfile,"w");
  while((c=fgetc(source))!=EOF)
  {
  if(isupper(c))
    c=tolower(c);
  else
```



```
c=toupper(c);
fputc(c,target);
}
fclose(source);
fclose(target);

source=fopen(sfile,"r");
printf("Displaying data from the SOURCE file:\n");
while(!feof(source))
{printf("%c",getc(source));
}
fclose(source);

target=fopen(tfile,"r");
printf("\nDisplaying data from the TARGET file:\n");
while(!feof(target))
{printf("%c",getc(target));
}
fclose(target);
getch();
}
```