OM VINAYAKA OM SAI RAM

INTRODUCTION TO C-LANGUAGE

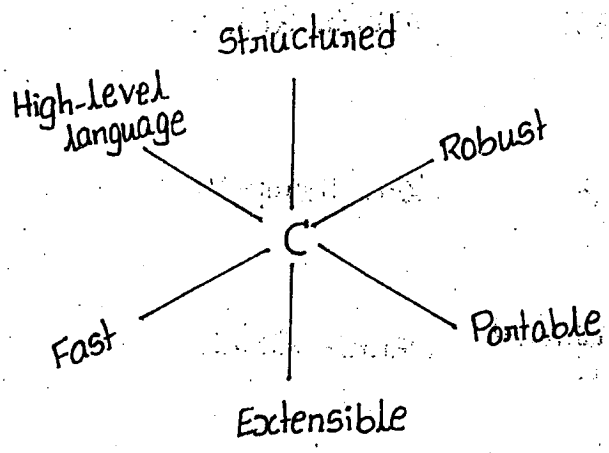
INTRODUCTION:-

⇒C is a powerful, portable and structured programming language.

⇒It combines the features of a high-level programming language with the elements of an assembly language, therefore it is suitable for writing both system software and application software.

⇒C has been used for implementing systems such as 1. Operating Systems
2. Compilers
3. Linkers
4. Word Processors
5. Utility Packages.

FEATURES OF C:-



⇒C is a robust language. It contains a rich set of built-in functions and operators, which can be used to write any complex programs

⇒C is highly portable. This means that C programs written for one computer can be run on another with little or no modification.

⇒Programs written in C are efficient and fast. This is due to variety of data types and powerful operators.

⇒ There are only 32 keywords in C.

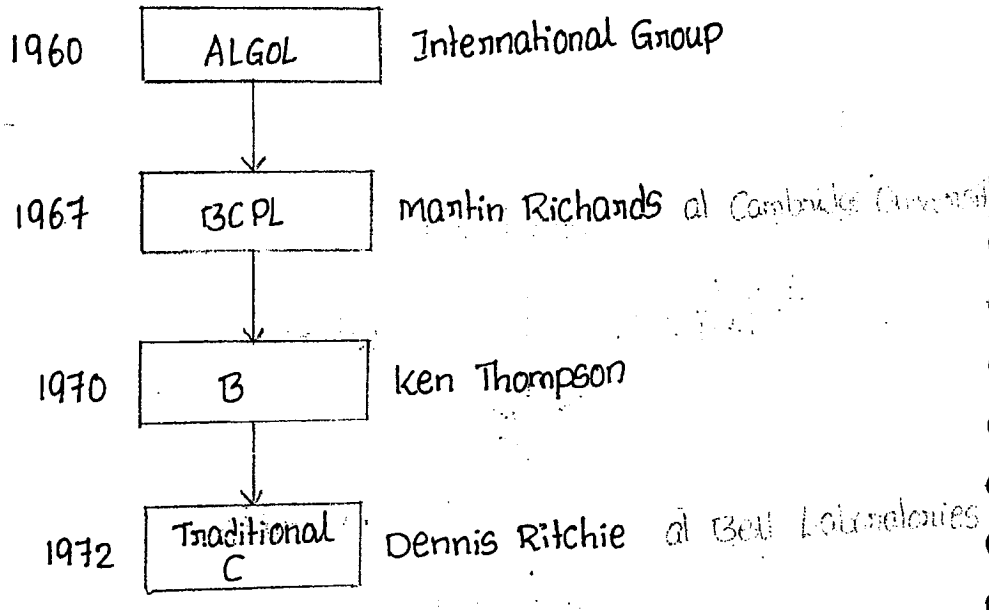
⇒ In C, several standard functions are available which can be used for developing programs.

⇒ C-language is well suited for structured programming.

⇒ C is extensible. Basically a 'C' program is a collection of functions that are supported by the C library. We can continuously add our own functions to C library.

HISTORY OF C-LANGUAGE:-

⇒ C is one of the most popular computer language today, because it is a structured, high-level, machine-independent language.



→ The root of all modern languages is ALGOL, introduced in 1960.

→ In 1967, Martin Richards developed a language called BCPL [Basic Combined Programming Language] for writing system software.

→ In 1970, Ken Thompson developed a language using many features of BCPL and named it as B. B was used to create early versions of UNIX operating system.

→ Finally in 1972, Dennis Ritchie developed C, which uses many concepts from ALGOL, BCPL, B and added the concept of data types and other powerful features.

→ C language was developed at Bell Laboratories in 1972.

→ The C language was developed along with the UNIX operating system. UNIX was coded entirely in C.

BASIC STRUCTURE OF C PROGRAMS:-

⇒ A C program can be viewed as a group of building blocks called functions.

⇒ A function is a set of statements designed to perform a specific task.

⇒ A C program may contain one or more sections as shown in the following figure.

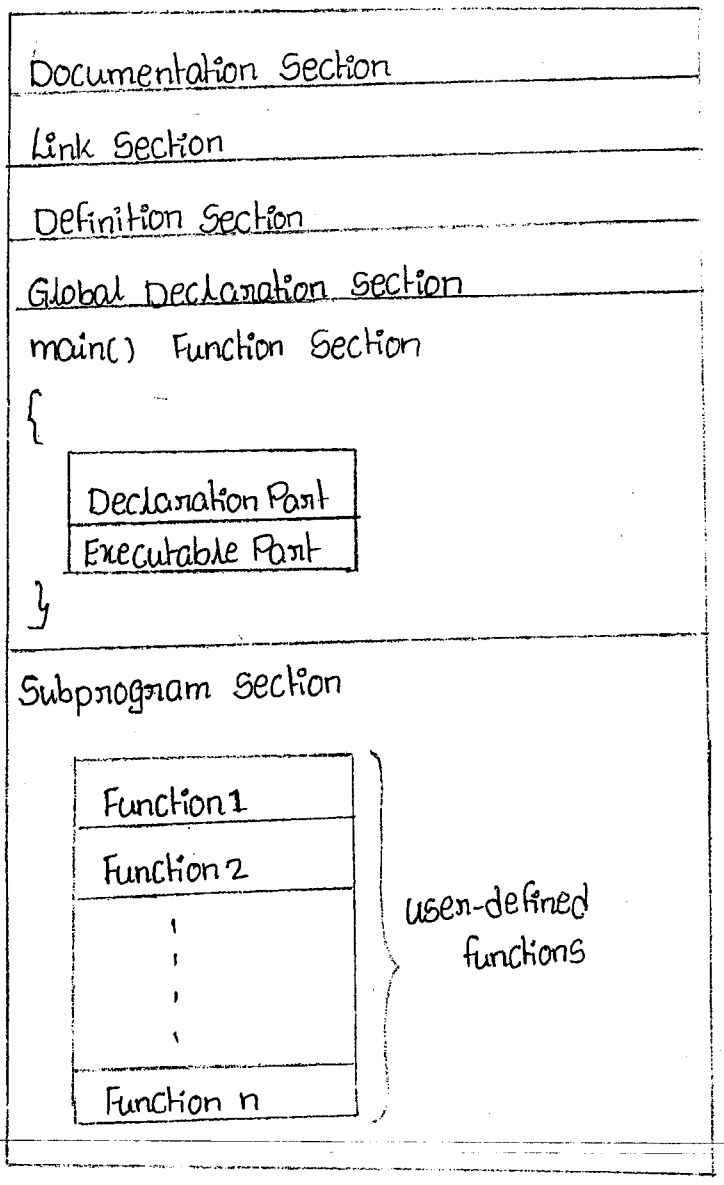


Figure: Structure of a C program

Documentation Section:-

⇒ The documentation section consists of a set of comment lines giving the name of the program, the author and other details.

⇒ The lines beginning with /* and ending with */ are known as comment lines.

⇒ These are used in a program to enhance its readability and understanding.

Example: /* Program to print Welcome message */

⇒ Comment lines are not executable statements, therefore anything between /* and */ is ignored by the compiler.

Link Section:-

⇒ The link section provides instructions to the compiler to link functions from C library.

⇒ C library consists of a set of functions, which are already predefined.

⇒ If we want to access the functions stored in the library, it is necessary to tell the compiler about the files to be accessed.

⇒ This is achieved by using the preprocessor directive #include.

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

Definition Section:-

⇒ The definition section defines all symbolic constants.

⇒ #define directive is used to define symbolic constants.

Ex: #define PI 3.14
#define MAX 100

Global Declaration Section:-

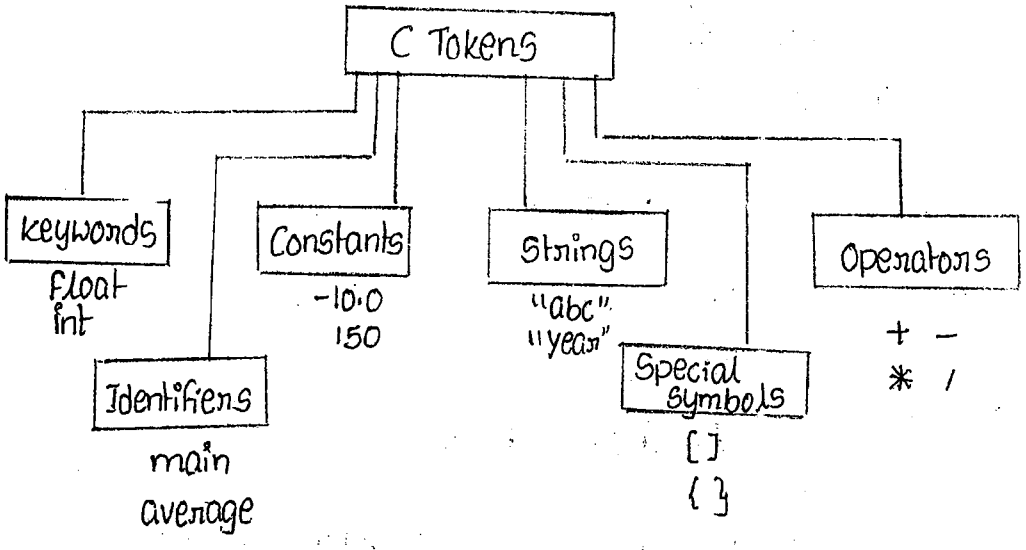
⇒ There are some variables that are used in more than one function. Such variables are called global variables and are declared in the global declaration section.
i.e. outside of all functions.

C TOKENS:-

⇒ In a passage of text, individual words and punctuation marks are called tokens.

⇒ In a C program the smallest individual units are known as C tokens.

⇒ C has six types of tokens.



⇒ C programs are written using these tokens and the syntax of the language.

KEYWORDS AND IDENTIFIERS:-

keywords:-

⇒ All keywords have fixed meanings and these meanings can't be changed.

⇒ C has 32 keywords.

Note: All keywords must be written in lowercase.

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

Identifiers:-

⇒ Identifiers refer to the names of variables, functions and arrays.

⇒ These are user-defined names and consists of a sequence of letters and digits, with a letter as first character.

⇒ Both uppercase and lowercase letters are permitted, but lowercase letters are commonly used.

⇒ The underscore(_) character is also permitted in identifier.

⇒ Rules for identifiers are as follows:

1. The first character must be an alphabet (or) underscore
2. It must consists only letters, digits or underscore.
3. Only first 31 characters are significant in an identifier.
4. It can't be same as keyword.
5. It must not contain white space.

Ex: average, max_marks, letter_1, etc.

⇒ Invalid Identifier — Reason

1Letter	— Begins with a digit
double	— keyword
TWO*FOUR	— Character * not allowed
Joe's	— Character ' not allowed.

CONSTANTS:-

⇒ Constants in C refers to the fixed values that do not change during the execution of a program.

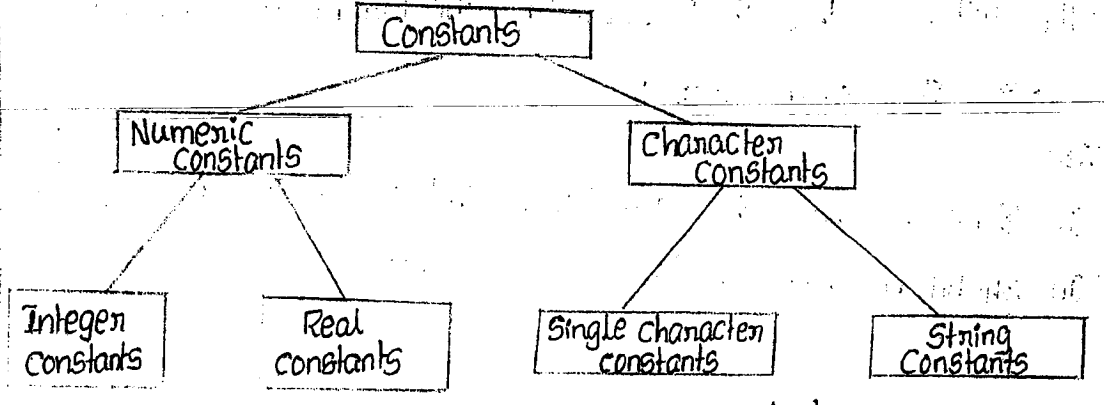


Figure: Basic types of C constants

Integer Constants:-

⇒ An integer constant refers to a sequence of digits.

⇒ There are 3 types of integers:

1. Decimal Integers
2. Octal Integers
3. Hexadecimal Integers.

1. Decimal Integers:-

⇒ Decimal integers consists of a set of digits, 0 through 9, preceded by an optional - (or) + sign.

Ex: 123 -321 0 654321 +78

Note: Embedded spaces, commas & special characters are not permitted between digits. [Ex: 20,000 $1000 are invalid].

2. Octal Integers:-

⇒ An octal integer constants consists any combination of digits from the set 0 through 7, with a leading 0.

Ex: 037 0 0435 0551

3. Hexadecimal Integers:-

⇒ A sequence of digits preceded by 0x (or) 0X are called as hexadecimal integers.

⇒ Hexadecimal integer constants consists of a set of digits 0 through 9 and alphabets A through F.

⇒ The letters A through F represents the numbers 10 through 15.

Ex: 0X2 0x9F 0X13CD

NOTE:-

⇒ In 16-bit machines the largest value that can be stored is 32767.

⇒ In 34-bit machines, it is 2,147,483,647.

Real Constants:-

⇒ Integer constants are inadequate to represent quantities that vary continuously, such as distances, heights, temparatures etc.

⇒ These quantities are represented by real numbers.

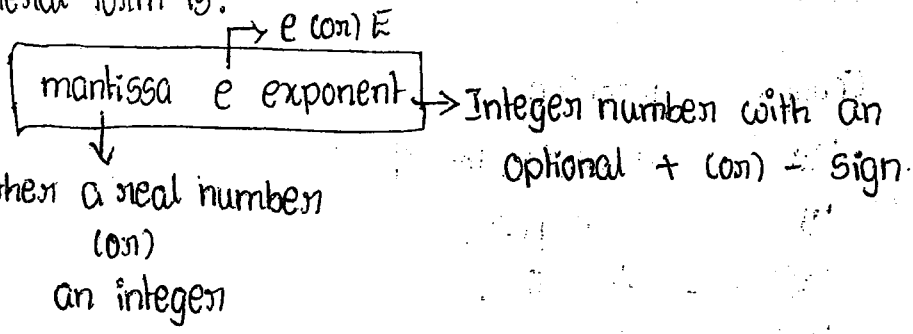
⇒ The numbers containing fractional part are called real constants.

Ex: 0.00083 -0.75 435.23 +247.0

⇒ These numbers are shown in decimal notation, having a whole number followed by a decimal point and the fractional part.

⇒ A real number may also be expressed in exponential [scientific] notation.

The general form is:



Example: The value 215.65 may be written as 2.1565e2 in exponential notation. e2 means multiple by 10^2.

⇒ Exponential notation is useful for representing numbers that are either very large or very small.

Ex: 750000000 may be written as 7.5E8 or 75E7.

-0.000000368 may be written as -3.68E-7

Single character constants:-

⇒ A single character constant contains a single character enclosed within a pair of single quote marks.

Ex: '5', 'x', 'P' etc.

⇒ Character constants have integer values known as ASCII values.

ASCII → American Standard Code for Information Interchanging.

Letters		ASCII values
0-9	⇔	48-57
A-Z	⇔	65-90
a-z	⇔	97-122

⇒ printf("%c", 'a'); → prints the letter a.

⇒ printf("%d", 'a'); → prints the number 97.

Backslash Character Constants:-

⇒ C supports some special backslash character constants that are used in output functions.

Constant	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

String Constants:-

⇒ A string constant is a sequence of characters enclosed in double quotes. The characters may be letters, numbers, special characters, and blank space.

Ex: "Hello!" "1987" "Well Done" "?--1" "X" "3+9" etc.

Note: 'x' ≠ "x"

VARIABLES:-

⇒ A variable is a data name that may be used to store a data value.

⇒ A variable value may change during program execution. i.e. A variable may take different values at different times during execution.

⇒ A variable name can be chosen by the programmer in a meaningful way so as to reflect its function.

Ex: Average, total, height, max_marks etc.

⇒ A variable name may consist letters, digits and underscore(_).

Rules:-

1. They must begin with a letter (or) Underscore(_).
2. ANSI standard recognizes a length of 31 characters. However, many compilers treat first eight characters are significant.
3. C compiler treats uppercase and lowercase letters as different. i.e. Total is not same as total (or) TOTAL
4. It shouldn't be a keyword.
5. White space is not allowed.

Special symbols:

Special symbols includes the following symbols

Symbol	Meaning		Symbol	Meaning
1. ~	Tilde	8.	"	Double quote
2. ,	Comma	9.	!	Exclamatory mark
3. .	period	10.	(	left parenthesis
4. ;	Semicolon	11.	)	Right "
5. :	colon	12.	[	left bracket
6. ?	Questionmark	13	]	Right "
7. '	single quote	14	{	left brace
		15	}	Right "

/, \, <, >, =, +, -, &, #, $, *, %, ^, _ ...

DATA TYPES:-

⇒ Data type refers to the type of data used in the program

⇒ C language is rich in its data types

⇒ The variety of data types available in C, allow the programmer to select the appropriate data type as per the need of the application.

⇒ Data types are mainly divided into 4 types:

1. Primary (or) Fundamental data types
2. Derived data types
3. User-defined data types
4. Empty data set

PRIMARY DATA TYPES:-

⇒ All C compilers support 4 fundamental data types:

1. Integer data type (int)
2. Character data type (char)
3. Floating point data type (float)
4. Double-precision floating point (double)

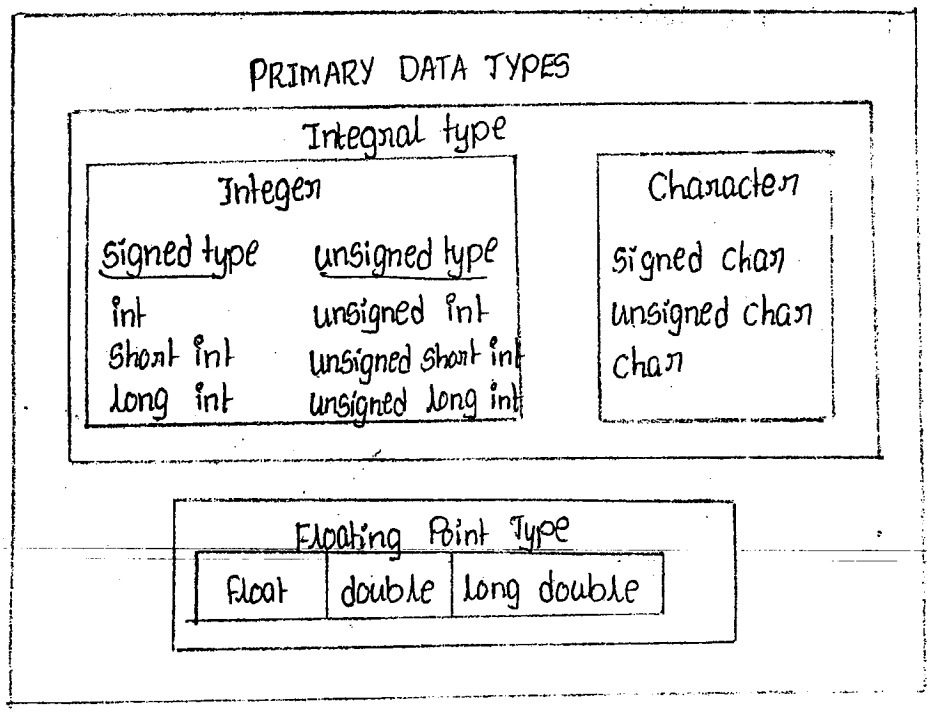


Figure: Primary data types in C.

INTEGER DATA TYPE:-

⇒ The int type is used to represent integers in C.

⇒ The keyword is int.

⇒ The size of int is 2 bytes.

⇒ The control string is %d

⇒ The range of int data type is: -32,768 to 32,767

⇒ In order to provide some control over the range of numbers, storage space, C has 3 classes of integers in both signed and unsigned forms.

1. short int → represents fairly small integer values
2. int
3. long int → large values, increases the range of values

TYPE	Size (bytes)	Range	Control String
int (or) signed int	2	-32,768 to 32,767	%d
unsigned int	2	0 to 65535	%u
short int (or) signed short int	1	-128 to 127	%d
unsigned short int	1	0 to 255	%u
long int (or) signed long int	4	-2,147,483,648 to 2,147,483,647	%ld
unsigned long int	4	0 to 4,294,967,295	%ud

⇒ short int represents fairly small integer values.

⇒ We declare long and unsigned integers to increase the range of values.

⇒ The use of qualifier 'signed' on integers is optional because by default declaration assumes a signed number.

Character TYPES:-

⇒ A single character can be defined as a character (char) type data.

⇒ Char occupies 1 byte of memory. i.e. characters are usually stored in 8 bits of internal storage.

⇒ Control string is: %c

⇒ Range is: -128 to 127

⇒ The qualifier signed (or) unsigned may be explicitly applied to char.

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

FLOATING POINT TYPES:-

⇒ A floating point (or) real number has an integral part and a fractional part, that are seperated by a decimal point.

⇒ Floating point numbers are defined in C by:

1. float data type
2. double data type

float:

→ Real numbers are stored in 32 bits, with 6 digits precision.

⇒ The size is 4 bytes, control string is: %f

→ When accuracy provided by float is not enough, the type double can be used to define a real number.

double:

→ A double type uses 64 bits giving a precision of 14 digits.

→ To extend the precision further we may use long double which uses 80 bits.

TYPE	Size	Range
float	4 bytes	3.4E-38 to 3.4E+38
double	8 bytes	1.7E-308 to 1.7E+308
long double	10 bytes	3.4E-4932 to 1.1E+4932

DERIVED DATA TYPES:-

⇒ The derived data types are: Arrays
Structures
Unions
Pointers

USER-DEFINED DATA TYPES:

⇒ C supports 2 user-defined data types

1. Type definition
2. Enumerated data type

1. Type definition [typedef]:-

⇒ Type definition allows users to define an identifier that would represent an existing data type.

⇒ The user-defined data type identifier can later be used to declare variables.

Syntax:- typedef type identifier → new name given to the data type
↓ keyword ↳ Existing data type

Examples: typedef int units;
typedef float marks;

→ Here unit symbolises int and marks symbolises float.
→ They can be used to declare the variables as follows:

units a, b, c;
marks sub1, sub2, sub3;

⇒ The main advantage of typedef is that we can create meaningful datatype names for increasing the readability of the program.

2. Enumerated data type [enum]:-

⇒ The enumerated data type can be defined as follows:

enum identifier {value1, value2, ------value n};
→ Enumeration constants

⇒ Here 'identifier' is a user-defined enumerated data type, which can be used to declare the variables, that can have one of the values enclosed within braces.

⇒ We can declare the variables as follows:

enum identifier V1, V2, ---Vn;

→ V1, V2, ----Vn can only have one of the values enclosed in braces.

Example: enum day {monday, Tuesday, ----, sunday};

enum day week-st, week-end;

week-st = monday;

week-end = sunday;

⇒ By default the compiler automatically assigns integer digits beginning with 0 to all enumeration constants.

i.e. monday is assigned with 0,

Tuesday is assigned with 1 and so on.

⇒ We can override the automatic assignments by assigning values explicitly to the enumeration constants.

Ex: enum day {monday=1, Tuesday, ----sunday};

EMPTY DATA SET:-

⇒ The void type has no values.

⇒ This is usually used to specify the type of functions.

⇒ The type of a function is said to be void when it doesn't return any value.

Ex: void main()

{

}

DECLARATION OF VARIABLES:-

Variable:- A variable is a data name which is used to store data values.

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

⇒ Variable declaration does two things:

1. It tells the Compiler what the variable name is.
2. It specifies what type of data the variable will hold.

Primary Type Declaration:-

⇒ A variable can be used to store a value of any data type.

⇒ The syntax for declaring variables is as follows:

```
data-type variable1, variable2, ------variableN;
```

→ Variables are seperated by commas.

→ A declaration statement must end with a semicolon.

Example:
```
int count;
double ratio;
int num1, num2, total;
char a,b,C;
```

⇒ The following program illustrates declaration of variables:

```
main
{
  /*------Declaration ----*/
  float x,y;
  int code;
  short int count;
  long int amount;
  unsigned int n;
  char C;
  double ratio;
  /* Computation */
  ----------
  ----------
  ----------
}
```

NOTE:
→ Declaration of variables is usually done immediately after the opening brace of the program.

User-defined type declaration:-

⇒ C supports 2 user-defined data types:

1. Type Definition
2. Enumerated Data type.

Type Definition:-

⇒ The general form is:

```
typedef type identifier;
```

→ The user-defined identifier can be used to declare the variables as follows:

```
identifier v1, v2, v3, --- vn;
```

Example:
```
typedef int units;
typedef float marks;
```

→ units, marks can be used to declare the variables as follows:

```
units batch1, batch2;
marks sub1, sub2;
```

→ Here batch1, batch2 are declared as int and sub1, sub2 are declared as float variables.

Enumerated Data type:-

→ The syntax is:

```
enum identifier {value1, value2, ---- value n};
enum identifier v1, v2, v3, --- vn;
```

Example:
```
enum day{monday, Tuesday, ---- sunday};
enum day week_st, week_end;
week_st = monday;
week_end = sunday;
```

MANAGING INPUT AND OUTPUT OPERATIONS:

⇒ There exist several functions for input and output operations in C.
⇒ These functions are collectively known as the standard I/O library.

1. Reading a character:

⇒ Reading a single character can be done by using the function getchar.
⇒ The getchar takes the following form:

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

→ Here, variable_name is a valid C name that has been declared as char type.
→ When this statement is encountered, the computer waits until a key is pressed and then assigns that character as a value to getchar function.

Example: char section;
section = getchar();

Example program:

```
/* The following example shows the use of getchar function */
#include<stdio.h>
main()
{
    char section;
    printf("Enter your section name:");
    section = getchar(); /* Reading a character */
    printf("your section is: %c", section);
    getch();
}
```

Character Test Functions:-

⇒ C supports several character functions, which are defined in the file ctype.h.

⇒ Each function takes single character as an argument.

Function	Test
isalnum(c)	— Is c an alphanumeric character?
isalpha(c)	— Is c an alphabetic character?
isdigit(c)	— Is c a digit?
islower(c)	— Is c lower case letter?
isupper(c)	— Is c an upper case letter?
isprint(c)	— Is c printable character?
ispunct(c)	— Is c a punctuation mark?
isspace(c)	— Is c an upper white space character.

Note:
We should include ctype.h in a program before using these functions

⇒ For example, isalpha(c) returns TRUE (non-zero value) if 'c' is an alphabet. Otherwise it returns FALSE (Zero).

Example program:-

→ Write a program that request the user to enter a character and displays a message telling that whether the entered character is an alphabet, or digit or any other special symbols.

```
#include<stdio.h>
#include<conio.h>
#include<ctype.h>
void main()
{
    char ch;
    clrscr();
    printf("Enter a character:");
    ch=getchar();
```

```
    if (isalpha(ch))
        printf("The entered character is an alphabet");
    else if (isdigit(ch))
        printf("The entered character is a digit");
    else
        printf("The entered character is a special symbol");
    getch();
}
```

Writing a character:-

⇒ putchar function is used for writing characters one at a time to the screen. i.e. putchar function displays one character at a time.

⇒ Syntax is:

```
putchar(variable-name);
```

→ Here variable-name is a char variable containing a character. This function displays the character contained in variable-name at the screen.

Example:
```
char answer;
answer = 'y';
putchar(answer); ⇒ Display the character Y on the screen.
```

Example Program:-

```
/* Write a program to read and display a character using getchar & putchar functions */

#include<stdio.h>
void main()
{
    char c;
    printf("Enter a character:");
    c=getchar(); /* Reading a character */
    printf("The entered character is:");
    putchar(c); /* Writing a character */
    getch();
}
```

Formatted Input:

⇒ Formatted input refers to the input data that have been arranged in a particular format.

Example: 15.75 123 John.

⇒ The scanf function is used to read formatted input.

⇒ The general form of scanf is:

```
scanf("Control String", &variable1, &variable2, ----);
```

→ The control string contains the format of data being received.

→ variable1, variable2, ---- are the variable names.

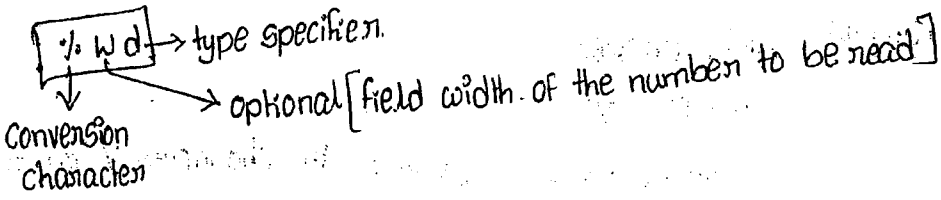
→ The & symbol before each variable name is an operator that specifies the variable address.

Note:

* We must always use &. Otherwise unexpected results may occur.

⇒ Control string contains:

→ Field (or format) specifications, conversion character %, type specifier and an optional number specifying the field width.



1. Inputting Integer numbers:-

⇒ The field specification for reading an integer number is %wd.

Example: scanf("%d", &number); ⇒ Reads an integer value from the keyboard.

Example 2:-

scanf("%2d", &num);

→ Suppose the following data are entered at the console:

31426

Now, the variable num will be assigned to 31 because of %2d.

Example 3:-

⇒ scanf("%2d %5d", &a, &b);

Suppose the input data are as follows:

31426 50

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

Example 4:-

⇒ What happens if we enter a floating point number instead of an integer?

Ans) The fractional part may be stripped away and also the scanf may skip reading further input.

Ex: scanf("%d", &a);

Suppose if we enter input data 34.56, the a value will be assigned to 34 [i.e. the scanf skips the fractional part].

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

→ suppose if we enter input data as follows:

23.567 6

→ The variable a will be assigned to 23. The scanf skips reading further input i.e. it doesn't read b value.

Example 5:-

⇒ An input field may be skipped by specifying * in the place of field width.

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

→ Suppose we enter the following data as input:

123 456 789

In this case a=123, b=789. The value 456 will be skipped (because of *).

2. Inputting Real Numbers:-

⇒ scanf reads real numbers using the specification %f.

Ex: scanf("%f %f %f", &x, &y, &z);

⇒ If we want to read double type numbers, then specification should be %lf

3. Inputting character Strings:-

⇒ scanf reads a single character using the specification %c

⇒ scanf can also read strings containing more than one character by using %s specification

4. Inputting Mixed data types:-

⇒ It is possible to use one scanf statement to input a data line containing mixed mode data.

Ex: scanf("%d %c %f %s", &count, &code, &ratio, &name);

Important Note:-

⇒ When a scanf function completes reading its list, it returns the value of number of items that are successfully read.

Ex: a = scanf("%d %c %f", &a, &b, &c);

→ Suppose the following data is entered at the console:

20 A 10.15

The a value will be assigned to 3. Since, the scanf reads 3 input values, and returns 3.

→ Considered the following data is entered:

20 10.15 A

scanf returns 1 and a value will be assigned to 1. Because the function encounter a floating point, when it was expecting a character. Therefore scanf terminates its scan after reading the first value.

FORMATTED OUTPUT:-

⇒ Output can be displayed by using printf function

⇒ The general form of printf statement is:

```
printf("Control String", arg1, arg2, ....., argn);
```

→ Where the control string may contain:

1. Characters that will be printed on the screen as they appear.

   Ex: printf("Welcome");

   The above printf function displays the message Welcome on the screen.

2. Format specifications that define the output format for display of each item.

   Ex: printf("%d", a);
   ↳ Format specifier

3. Escape sequence characters such as \n, \t and so on.

   Ex: printf("\n SITAMS \n CHITTOOR");

→ The arguments arg1, arg2, ... argn are the variables whose values are printed according to the format specifications.

Examples:-

1. printf("Programming in C"); → Displays a message Programming in C
2. printf(" "); → Displays one space
3. printf("\n"); → New line
4. printf("%d", x); → Display the value of x
5. printf("a=%f \n b=%f", a, b); → Display the values of a, b
6. printf("\n \n");
7. printf("sum = %d", 2+3); → Display sum = 5

Output of Integer Numbers:-

⇒ The format specification for printing an integer number is %d

Ex: int a=10;
printf("%d", a); → Display 10
printf("a=%d", a); → Display a=10

Ex: int a,b;
a=10;
b=20;
printf("a+b=%d", a+b); → Display a+b=30
printf("Sum=%d", a+b); → Display Sum=30

Outputting Real Numbers:-

⇒ The format specifier for printing a real number is %f

Ex: float a=5.7;
printf("%f", a); → Display 5.7

Outputting a Single Character:-

⇒ A single character can be displayed by using the %c format specifier.

Ex: char a='x';
printf("%c", a); → Display x

Mixed Data Output:-

⇒ It is possible to mix data types in one printf statement.

Ex: int a=100;
char b='z';
float c=10.57;
printf("%d IE %C IE %f", a,b,c); → Display 100 z 10.57

OPERATORS & EXPRESSIONS

⇒ In order to perform different kinds of operations, C uses different operators.

⇒ An operator is a symbol that tells the computer to perform certain mathematical or logical manipulations.

⇒ An operand is a data item on which operators perform the operations

Ex: 5 + 3 → operands
↳ operator

⇒ Operators are used in program to manipulate data and variables.

⇒ C operators are classified into:
1. Arithmetic operators (+, -, *, /, %)
2. Relational operators (<, >, <=, >=, ==, !=)
3. Logical operators (&&, ||, !)
4. Assignment operators (=)
5. Increment and decrement (++, --) operators
6. Conditional operators (? :)
7. Bitwise operators (&, |, ^)
8. Special operators (,, sizeof, &)

Expression:

An expression is a sequence of operands and operators that reduces to a single value.

Ex: x = 5*4 + 8/2;

ARITHMETIC OPERATORS :-

⇒ C provides all the basic arithmetic operators.

Operator	Meaning	Example
+	Addition (or) Unary plus	2+2 = 4
−	Subtraction (or) Unary minus	5-3 = 2, -3, -4
*	Multiplication	5*2 = 10
/	Division	10/2 = 5
%	Modulo division	11%3 = 2 (Remainder)

⇒ The above operators can operate on any built-in data type allowed in C.

⇒ The unary minus operator, multiplies its sigle operand by -1.

⇒ Integer division truncates any fractional part.

Ex: 14/4 = 3 (decimal part truncated).

⇒ Modulo division produces the remainder of an integer division.

Ex: 14%4 = 2 (Remainder of division)

```
   3
4) 14
   12
   ---
   (2) → Remainder
```

> **Note:** The modulo division operator % can't be operated on floating point data.

Integer Arithmetic :-

⇒ An arithmetic operation involving only integer operands is called integer arithmetic and the expression is called integer expression.

⇒ Integer arithmetic always yields an integer value.

Ex: a = 14, b = 4

a-b = 10
a+b = 18
a*b = 56
a/b = 3
a%b = 2

⇒ During modulo division, the sign of the result is always the sign of the first operand.

Ex: -14 % 3 = -2
-14 % -3 = -2
14 % -3 = 2

Real Arithmetic:-

⇒ An arithmetic operation involving only real operands is called real arithmetic.

⇒ The result is always floating point value.

Ex: x = 6.0/7.0 = 0.857143
z = -2.0/3.0 = -0.666667

Mixed-mode Arithmetic:-

⇒ When one of the operands is real and other is integer, then it is called mixed-mode arithmetic.

⇒ If either operand is of the real type, then the result is always a real number.

Ex: 15/10.0 = 1.5

RELATIONAL OPERATORS:-

⇒ Relational operators are used to compare two quantities, and depending on their relation, we take certain decisions

⇒ An expression containing relational operator is known as relational expression.

Ex: a<b (or) 1<20

⇒ If the relation is true, it returns 1. Otherwise 0 for false relation.

⇒ C supports six relational operators.

Operator	meaning	Example	Return value
>	Greater than	5>4	1
>=	Greater than or equal to	11>=5	1
<	Less than	10<9	0
<=	Less than or equal to	10<=10	1
==	equal to	2==3	0
!=	Not equal to	3!=3	0

⇒ Relational expressions are used in decision statements such as if, while, do-while, for.

LOGICAL OPERATORS:-

⇒ The logical operators are used when we want to test more than one condition and make decisions.

⇒ C supports 3 logical operators.

Operator	meaning	Example	Return value
&&	– Logical AND	5>3 && 5<10	0 or 1
||	– Logical OR	2==3 || 4==4	1
!	– Logical NOT	!(2==3)	1

⇒ An expression which combines two or more relational expressions is called logical expression or a compound expression

Ex: 5>3 && 5<10

⇒ The logical expression yields 1 or 0 according to the truth table.

⇒ AND Truth table:

Condition1	Condition2	Return value
1	1	1
1	0	0
0	1	0
0	0	0

⇒ The logical AND (&&) operator provides true only when both the expressions are true, otherwise 0.

⇒ Logical OR Truth table:-

Condition1	Condition2	Return value
1	1	1
1	0	1
0	1	1
1	1	0

⇒ The logical OR (||) provides true when one of the expression is true, Otherwise 0

⇒ The logical NOT (!) provides 0 if the condition is true, otherwise 1.

Examples	Return value
5>3 && 5<10	1
8>5 || 8<2	1
!(8==8)	0

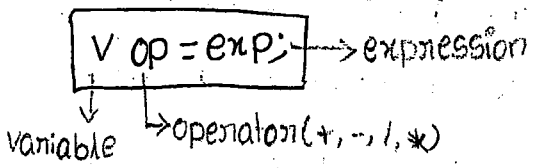
ASSIGNMENT OPERATORS:-

⇒ Assignment operators are used to assign the value or result of an expression to a variable.

Syntax: `Variable_name = constant (OR) expression`

Example: a=10 , c=20
b=10*5

⇒ In addition, C has set of 'shorthand' operators of the form



→ The operator op= is known as shorthand assignment operator.

⇒ The assignment statement V op=exp is equivalent to:

`V = V op (exp);`

⇒ For example x+=3 is equivalent to x=x+3

x+=y+1 is equivalent to x=x+(y+1)

⇒ Some of the commonly used shorthand assignment operators are:

Statement with simple assignment operator	Statement with shorthand operator
a=a+1	a+=1
a=a-1	a-=1
a=a*(n+1)	a*=(n+1)
a=a/(n+1)	a/=n+1
a=a%b	a%=b

⇒The use of shorthand assignment operators has 3 advantages:

1. What appears on the left-hand side need not be repeated
2. The statement is easier to read.
3. The statement is more efficient.

INCREMENT AND DECREMENT OPERATORS:-

⇒ C has two very useful operators: increment and decrement operators

operator	meaning
++	Increment
--	Decrement

⇒The operator ++ adds 1 to the operand.

i.e. x++ is equivalent to x=x+1

⇒ The operator -- subtracts 1 from its operand.

i.e. x-- is equivalent to x=x-1.

⇒Both these operators may either follow or precede the operand.

i.e. x=x+1 can be represented as x++ (or) ++x

Post Increment/Decrement:-

⇒ If ++ or -- are used as a suffix to the variables name then post increased/decreased operations take place.

Ex: m=5;
y=m++;

→ After executing these statements y would be 5 and m=6.

→ i.e. The postfix operator first assigns the value to the variable on lefthand side and then increments the operand.

Ex: x=20;
y=x--;

Here after executing y=20, x=19

Pre Increment/Decrement :-

⇒ If "++" or "--" are used as a prefix to the variable name then pre increment/decrement operations take place.

Ex: m=5;
y=++m;

→ After executing these statements, y=6, m=6. i.e. A prefix operator first adds 1 to the operand and then the result is assigned to the variable on left.

Ex: m=5;
y=--m;

→ In this case the value of m and y would be 4.

⇒ Note: We use increment and decrement statements in for and while loops extensively.

Examples:

x=10, y=20	x=10, y=20
z=x*y++ ⇒ z=200	z=x*++y ⇒ z=210
a=x*y ⇒ a=210	a=x*y ⇒ a=210

NOTE:-

⇒ When postfix ++ (or --) is used with a variable in an expression, the expression is evaluated using the original value of the variable and then the variable is 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.

CONDITIONAL OPERATOR:-

⇒ The operator '?:' is known as conditional operator.

⇒ The conditional operator has the following form:

```
condition ? statement 1 : statement 2;
```

Working:-

1. First the condition is evaluated.
2. If the condition is true, then statement1 will be executed.
3. If the condition is false, then statement 2 will be executed.

⇒ The operator '?:' is also known as ternary operator. Because it takes 3 operands.

Ex:
```
a=10;
b=15;
x=(a>b)?a:b;
```

Here x will be assigned to 15. Because first it checks the condition a>b (i.e. 10>15), condition is false. So statement2 will be executed. So b is assigned to x.

Ex:
```
a=25;
b=10;
x=(a>b)?a:b;
```

Here x will be assigned to 25 (∵ a>b is true).

```
(a>b)? printf("%d", a) : printf("%d", b);
printf("%d", a>b ? a : b);
```

BITWISE OPERATORS:-

⇒C supports bitwise operators for manipulation of data at bit level.

⇒These operators are used for testing the bits, on shifting them right on left.

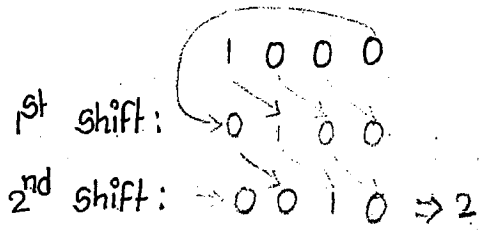
⇒These operators can operate only on integer data.

Operator	Meaning
>>	Right shift
<<	Left shift
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
~	One's complement.

Right Shift (>>):

Example: Shift the number 8 by 2 bits right.

Sol: 8 binary equivalent 1000



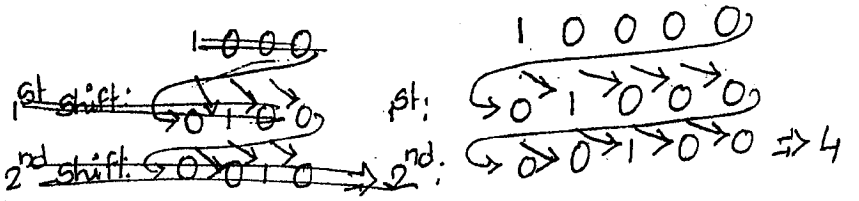
⇒ Shifting two bits right means, the inputted number is to be divided by 2^s, where s is no.of shifts.

i.e. $y = n/2^s$

where n = Number
s = Number of positions to be shifted.

Example2: Shift 16 by 2 bits right

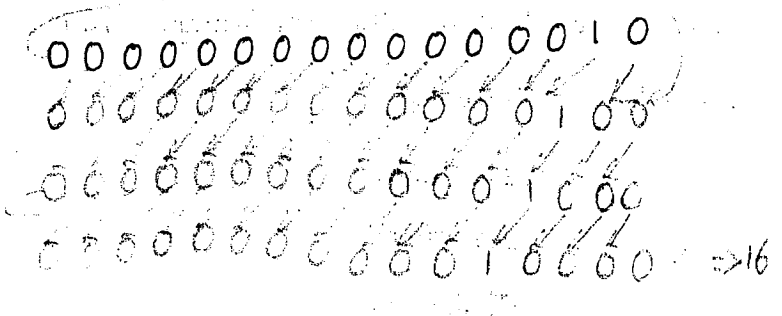
16 binary equivalent 10000



Left Shift (<<):-

Example: Shift the number 2 by 3 bits left

2 binary equivalent



⇒ Shifting 3 bits left means, multiply the number by 2^3.

i.e. $y = n * 2^3$.

Bitwise AND:-

Example: consider a=8 and b=4

c = a & b

a=8 : 1 0 0 0

b=4 : 0 1 0 0

c=a&b : 0 0 0 0

∴ c=0

a=8, b=8

c = a & b

a : 1 0 0 0

b : 1 0 0 0

c : 1 0 0 0

∴ c=8.

Bitwise OR:-

Example: Consider a=8 and b=4

c = a | b

a : 1 0 0 0

b : 0 1 0 0

c=a|b : 1 1 0 0

∴ c=12

Bitwise Exclusive OR:-

Consider a=8 and b=2

```
c = a^b;

a:   1 0 0 0
b:   0 0 1 0
c=a^b: 1 0 1 0

∴ c = 10
```

TRUTH Table of XOR

X	Y	Z
1	1	0
0	1	1
1	0	1
0	0	0

One's Complement:-

Input	OUTPUT
1	0
0	1

Example: consider a=8

```
c = ~a

a:   1 0 0 0
~a:  0 1 1 1  ⇒ 7
```

SPECIAL OPERATORS:-

⇒ C supports the following special operators:

1. Comma Operator
2. Sizeof operator
3. & operator

1. Comma operator:-

⇒ The comma operator is used to seperate two or more expressions.

⇒ A comma-linked list of expressions are evaluated left to right and the value of the right-most expression is the value of the combined expression.

Ex: a=2,4,5 ⇒ a=5

value = (x=10, y=5, x+y) ⇒ value = 15

(x=x+y, y+1)

⇒ Some applications of comma operator:

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

Exchanging values: t=x, x=y, y=t;

2. sizeof operator:-

⇒ The sizeof operator gives the number of bytes occupied by a operand.

→ The operand may be a variable, a constant or a data type qualifier.

Ex: int sum;
float avg;

sizeof(sum) → Returns 2
sizeof(avg) → Returns 4

Ex: m = sizeof(sum);
n = sizeof(double);
k = sizeof(235L);

3. & operator:-

⇒ The & operator prints the address of the variable in the memory.

Ex:
```
#include<stdio.h>
void main()
{
    int a;
    a=20;
    printf("%d", a); /* Displays value of a */
    printf("%u", &a); /* Displays address of variable a */
}
```

⇒ Write a program to display number of bytes occupied by different data types

```
#include<stdio.h>
void main()
{
    printf("Number of bytes occupied by int is: %d", sizeof(int));
    printf("Number of bytes occupied by char is: %d", sizeof(char));
    printf("Number of bytes occupied by float is: %d", sizeof(float));
    printf("Number of bytes occupied by double is: %d", sizeof(double));
}
```

OPERATOR PRECEDENCE AND ASSOCIATIVITY:-

⇒ Every operator has a precedence value.

⇒ Precedence of operators refers to the order in which they are operated in a program (or) expression.

⇒ Associativity specifies the order in which the operators are evaluated when they have same precedence.

→ Associativity is of two types:

1. Left-to-Right Associativity: Evaluates an expression starting from left and moving towards right.

2. Right-to-Left Associativity: Evaluates from right to left.

⇒ The precedence and associativity of various operators in C are as shown in the following table:

Operator	Description	Associativity	Precedence
( )	Function call	Left to Right	1
[ ]	Array element Reference		
+	Unary plus	Right to Left	2
−	Unary minus		
++	Increment		
−−	Decrement		
!	Logical negation		
~	One's complement		
*	Pointer reference		
&	Address		
Sizeof	Size of an object		
(type)	Type cast (conversion)		
*	Multiplication	Left to Right	3
/	Division		
%	Modulo division		
+	Addition	Left to Right	4
−	Subtraction		
<<	Left Shift	Left to Right	5
>>	Right Shift		

EXPRESSIONS:-

An expression is a sequence of operands and operators that reduces to a single value.

Ex: x = 5*4+8/2;

c = a+b;

⇒ Based on the position and number of operands, expressions are classified into 6 types.

1. PRIMARY EXPRESSION:-

⇒ An ordinary mathematical expression is called infix expression.

⇒ In infix, operators are placed between the operands in an expression.

Ex: (A/B)+(C-D)*E

2. POSTFIX EXPRESSION:-

⇒ In postfix expressions the operators are placed after the operands

Ex: Function call → Clrscr()

a++

a--

3. PREFIX EXPRESSION:-

⇒ In prefix expressions, the operators are placed before the operands.

Ex: ++a

--a

4. UNARY EXPRESSION:-

⇒ An unary expression is like a prefix expression consists of one operand and one operator and the operand comes after the operator.

Ex: ++a, -b, +d

5. BINARY EXPRESSION:-

⇒ It is a combination of 2 operands and an operator.

Ex: a+b, c*d

6. TERNARY EXPRESSION:-

⇒ It is an expression which consists of a ternary operator pair "?:"

Ex: condition ? exp1 : exp2;

TYPE CONVERSIONS IN EXPRESSIONS:-

⇒ C permits mixing of constants and variables of different types in an expression.

⇒ C performs conversions between different data types. i.e. it converts one data type into another data type.

⇒ There are two types: 1. Implicit-type Conversion
2. Explicit-type conversion.

Implicit type Conversion:-

→ Automatic type conversion by the compiler.

→ C automatically converts any intermediate values to the proper type. This automatic conversion is known as implicit type conversion.

RULES:-

→ During the expression evaluation, if the operands are of different data types, the 'lower' type is automatically converted to the 'higher' type before the operation proceeds.

Ex: 3+1.5 = 4.5

a+10 = 107 ⇒ Here a is char. It is converted into integer. Ascci equivalent of a is 97. so 97+10=107.

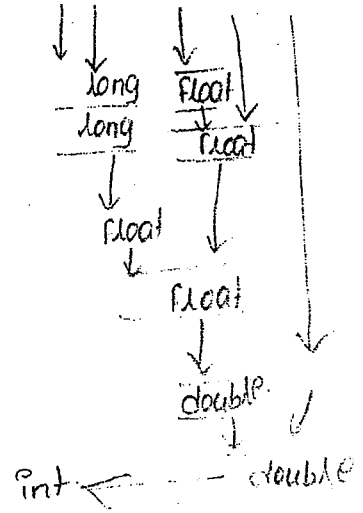
→ In general

char < int < unsigned int < long int < unsigned long int < float < double < long double

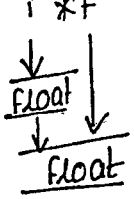
⇒ For assignment statements, C converts the value at the right side of the assignment statement to the type of the variable on the left side.

Ex:
```
int a;
Char ch;
Float f;
Ch=a; /* Here a is integer that is converted to char */
a=f; → float is converted into int
f=ch; → Char is converted into float.
f=a; → int is converted into float.
```

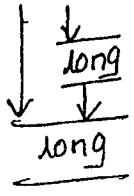
Ex:
```
int i,x;
float f;
double d;
long int l;
x = l/i + i*f - d
```



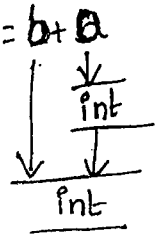
Ex:
```
int i;
float f;
i*f
```
Here i is integer, f is float. During compilation lower type is converted into higher type. i.e. int is converted into float.



Ex:
```
int i;
long int x,l;
x=l/i;
```



Ex:
```
Char a='A';
int b=10;
c=b+a
```



c = 10+65
= 75

Explicit Type Conversion:-

→ The type of an expression can also be converted explicitly by type casting.

Syntax: `(type-name) expression` → Here expression is converted into the data type specified in paranthesis.

Ex: `x = (int) 7.5` → 7.5 is converted to integer by truncation.

`a = (int) 21.3 / (int) 4.5` → Evaluated as 21/4

`b = (double) sum/n` → Division is done in floating point mode.

`y = (int) (a+b)` → The result of a+b is converted to integer.

`z = (int) a+b` → a is converted to integer and then added to b.

DECISION MAKING AND BRANCHING / SELECTION

⇒ Flow of control:- Flow of control shows the order in which the program is executing.

⇒ Normally a C program is a set of statements executed in sequence. i.e. All the statements are executed sequentially in the order in which they appear.

⇒ But in pratical, there are many situations where we have to change the order of execution of statements based on certain conditions.

⇒ This involves decision-making to see whether a particular condition is satisfied or not. Based on the condition the order of execution of statements will be changed.

⇒ C language supports the following decision-making statements.

1. if statement
2. switch statement
3. goto statement.

→ These statements control the flow of execution, so they are also called as control statements.

Decision making with if statement:-

⇒ The if statement is a powerful decision-making statement and is used to control the flow of execution of statements.

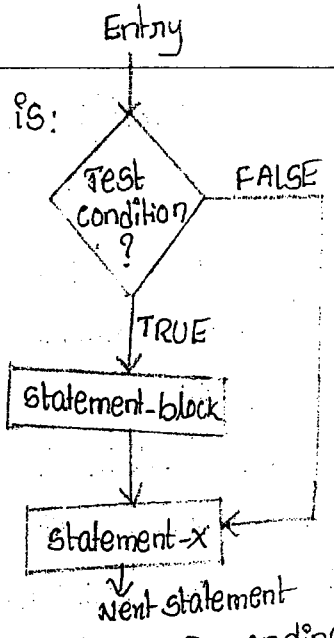
⇒ The if statement has the following forms:

1. Simple if statement
2. If-else statement
3. Nested if...else statement
4. else if ladder.

Simple if statement:-

⇒ The general form of simple if statement is:

```
if( test-condition)
{
    statement-block;
}
 statement-x;
```



WORKING:-

⇒ First the computer will evaluate the test condition. Depending on whether the condition is true or false, it transfers the control to a particular point.

⇒ The statement-block may be a single statement or a group of statements.

⇒ If the condition is true, the statement-block will be executed. If the condition is false, it will skip the statement-block and control will jump to statement-x.

Example Programs:-

① Write a program to check the equivalence of two numbers using simple if statement.

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

Explanation:-

⇒ In this program two numbers a & b are entered.

⇒ Their equivalence is checked by using the condition if(a==b)

⇒ If condition is true it will display the message "both numbers are equal"

⇒ If the condition is false, it will not do any thing.

(2) Write a program to perform the ratio of a/b and print the result when b is not equal to zero.

```
#include<stdio.h>
Void main()
{
   int a,b;
   float ratio;
   printf("Enter a,b values:");
   scanf("%d %d",&a,&b);
   if(b!=0)
   {
      ratio = a/b;
      printf("Ratio = %f", ration);
   }
}
```

Sample output:-

Enter a,b values 3 2

Ratio = 1.5

Sample output:-

Enter a,b values 5 0

/* Nothing will be displayed */

Explanation:-

⇒ In the above program, the values of a,b are read from keyboard.

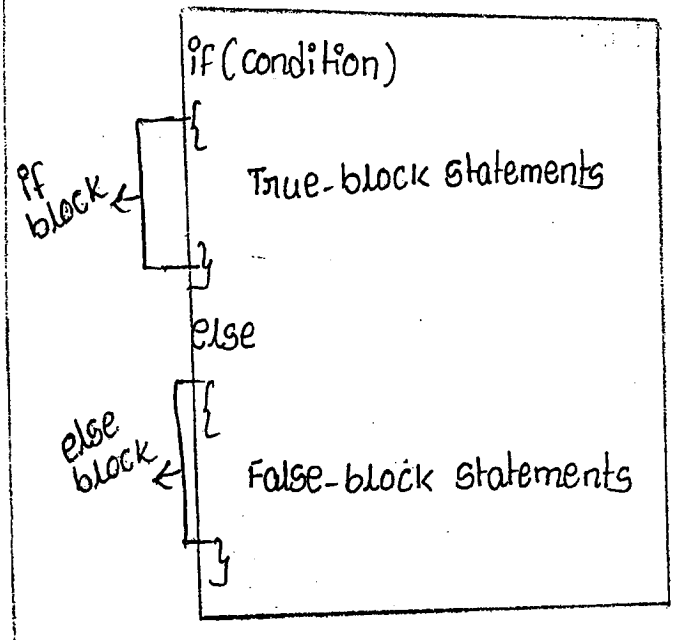
⇒ It will check the condition whether the value of b is equal to 0 or not.

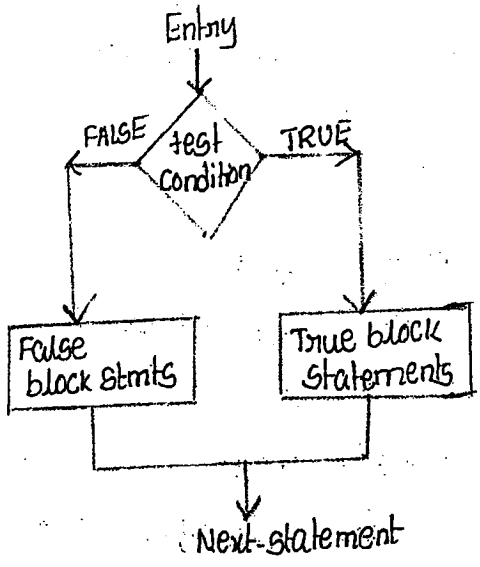
⇒ If b!=0 then it will calculate the ratio and print the result.

⇒ If b value is 0, then the program will not do anything.

The if-else statement:-

⇒ The if-else statement is an extension of simple if statement.
⇒ The simple if statement does nothing when the condition is false.
⇒ The if-else statements takes care of both true as well as false conditions.
⇒ It has two blocks.
→ One block is for "if" and is executed when the condition is true.
→ Another block is executed when the condition is false.
⇒ The general form of if-else statement is as follows:





WORKING:-
→ First it will check the condition.
→ If the condition is true, if block statements (i.e. true-block) statements will be executed.
→ If the condition is false, then else block statements (i.e. false-block) statements will be executed.
→ In either case, either true or false block statements will be executed. But not both.

Important Note:-
⇒ The 'else' statement can't be used without 'if' statement.
⇒ No multiple else statements are allowed with one if statement.

Example program:-

① Write a program to find biggest of 3 numbers using nested if-else.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a,b,c;
    clrscr();
    printf("Enter a,b,c values:");
    scanf("%d %d %d",&a,&b,&c);
    if(a>b)
    {
        if(a>c)
        {
          printf("%d is biggest number",a);
        }
        else
        {
          printf("%d is biggest number",c);
        }
    }
    else
    {
        if(c>b)
        {
          printf("%d is biggest number",c);
        }
        else
        {
          printf("%d is the biggest number",b);
        }
    }
    getch();
}
```

The else-if ladder:-

⇒ When multiple decisions are involved, then else-if ladder is used.

⇒ The general syntax of else-if ladder is as follows:-

```
if(condition-1)
{
   statement-1;
}
else if(condition-2)
{
   statement-2;
}
else if(condition-3)
{
   statement-3;
}
.
.
.
else
{
   default-statement;
}
statement-x;
```

⇒ The conditions are checked from top to bottom.

⇒ If condition 1 is true then statement-1 will be executed.

⇒ If condition 1 is false, then the control is transferred to check the next condition.

⇒ If anyone of the condition is true, then the statements associated with it will be executed.

⇒ This process continues till the last.

WORKING:-

⇒ The conditions are evaluated from top to bottom.

⇒ As soon as a true condition is found, the statements associated with it is executed and the control is transferred to statement-x.

⇒ When all the n conditions becomes false, then the final else containing default-statement will be executed.

Example program:-

⇒Write a program to calculate commission based on the sales amount.

Sales amount	Commission
< 5000	NIL
>=5000 && <10000	2% of sales amount
>=10000	5% of sales amount

```
#include<stdio.h>
#include<conio.h>
void main()
{
    float sa, cm;
    clrscr();
    printf("Enter sales amount:");
    scanf("%f", &sa);
    if(sa<5000)
    {
      printf("Commission is NIL");
    }
    else if(sa>=5000 && sa<10000)
    {
        cm = 0.02*sa;
        printf("Commission is: %f", cm);
    }
    else
    {
        cm=0.05*sa;
        printf("Commission is: %f", cm);
    }
    getch();
}
```

The Switch Statement:-

⇒ The switch statement is a multi-way branch statement.

⇒ If there is a possibility to make a choice from a number of options, then switch statement is used.

⇒ The switch statement requires only one argument of any data type.

⇒ The switch statement checks the value of a given argument against a list of case values and when a match is found, a block of statements associated with that case is executed. If no match then the default statement will be executed.

⇒ The general form of switch statement is as follows:

```
switch(variable or expression)
{
   case value1 : block1;
                 break;

   case value2 : block2;
                 break;

   case value3 : block3;
                 break;

   --------
   --------

   default : default-block;
             break;
}
```

⇒ switch, case, default and break are keywords.

⇒ The break statement is used to exit from the current case structure. The break statement transfer the control out of the switch statement.

⇒ The expression is an integer expression or characters.

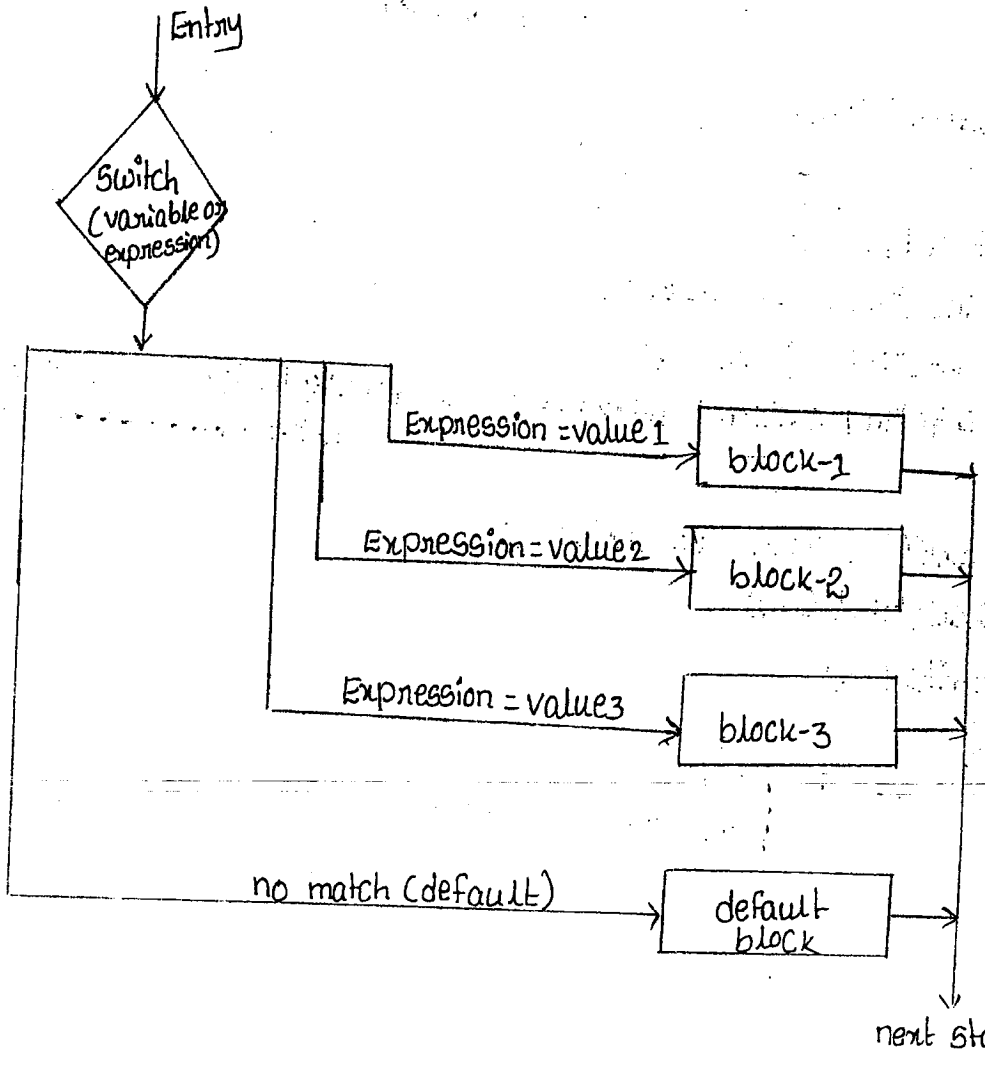
⇒ The values value1, value2, value3, ... are constants and are known as case labels. Each of these values should be unique within a switch statement.

⇒ Each case label should end with colon(:). Case lables may be integer numbers like 1,2,3. --- or it may be character constants like 'A','B','C',....

⇒ block1, block-2. ---- are statement lists and may contain zero or more statements.

⇒ When the switch is executed, the value of the expression is successfully compared against the case values. If a match is found, the block of statements associated with that case are executed.

⇒ The default is an optional case. If present, it will be executed if the value of the expression doesn't match with any of the case values.



Note:

→ The break statement is optional.

If break statement is not present, then all cases are executed.

→ There must be exactly one default statement, and it is optional.

→ case labels must be unique. No two labels can have the same value.

→ case labels must be constants.

Example program:-

→ Write a program to perform the arithematic operations based on the user choice.

Choice	operation
1	Addition
2	Subtraction
3	multiplication
4	Division
5	modulo division

```
#include<stdio.h>
main()
{
    int a,b,c,ch;
    printf("Enter a,b values:");
    scanf("%d %d", &a, &b);
    printf("\n 1. Addition \n 2. Subtraction \n 3. multiplication \n 4. Division
                                                \n 5. modulo division");

    printf("\n Enter your choice:");
    scanf("%d", &ch);
    switch(ch)
    {
        case 1: c=a+b;
                printf("\n Addition: %d", c);
                break;

        case 2: c=a-b;
                printf("\n Subtraction: %d", c);
                break;
```

```
        case 3: c=a*b;
                printf("Multiplication : %d", c);
                break;
        case 4: c=a/b;
                printf("Division : %d", c);
                break;
        case 5: c=a%b;
                printf("Modulo division: %d", c);
                break;
        default: printf("Wrong choice");
                break;
    }
    getch();
}
```

Sample output:-

```
Enter a,b values: 6 3
1. Addition
2. Subtraction
3. Multiplication
4. Division
5. modulo division
Enter your choice : 2
Subtraction : 3
```

Nested switch statement:-

⇒ We can place one switch statement inside another switch statement.

⇒ The inner switch() can be a part of the outer() switch.

⇒ The general form of nested-switch statement is as follows:

```
switch(expression)
{
    case 0:
        -----
        -----
        break;
    case 1: switch(expression)
            {
                case 0: block-1;
                        break;
                case 1: block-2;
                        break;
                  :
                default: staments;
                         break;
            }
        break;
    case 2: ------
            ------
            break;
      :
    default: default-statements;
             break;
}
```

Outer switch

Inner switch

⇒ The inner switch() and outer switch() labels may be same.

Example program:-

⇒ Write a program to determine whether the given number is even or odd. Use nested switch() statements.

```
#include<stdio.h>
#include<conio.h>
Void main()
{
   int x,y;
   Clrscr();
   printf("Enter x value:");
   scanf("%d",&x);
   switch(x)
   {
      case 0: printf("Number is Even");
              break;
      case 1: printf("Number is odd");
              break;
      default : y=x%2;
                switch(y)
                {
                   case 0: printf("Number is Even");
                           break;
                   case 1: printf("Number is odd");
                           break;
                }

   }
   getch();
}
```

Explanation:- In the above program the first switch is used for displaying a message such as even or odd when the entered number is 0 or 1.

→ When the number is other than 0 and 1, its remainder is calculated and stored in y. The variable y is used in the inner switch. If the remainder is 0, it will display the number is even. Otherwise it will display the number is odd.

The goto Statement:-

⇒ The goto statement is an unconditional brach statement. i.e. it doesn't require any condition.

⇒ The goto statement transfers the control from one part of a program to another part without testing any condition.

⇒ The goto statement has the following form:

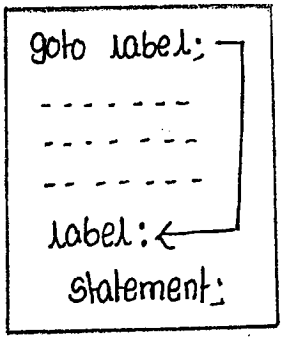
```
goto label;
```

where goto → is a keyword.

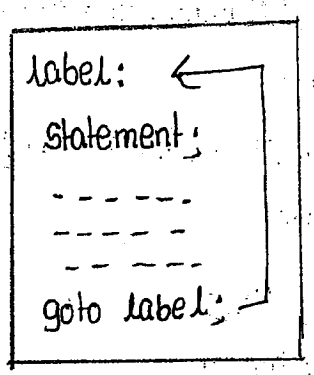
label → is the position where the control is to be transferred.

A label must be a valid variable name.

⇒ The general form of goto and label are as follows:



Forward jump



Backward jump.

⇒ The goto requires a label inorder to identify the place ~~whether~~ where the control is to be transferred.

⇒ The label: can be any where in the program either before on after goto label; statement.

⇒ During program execution, when a statement like goto label; is met, the control is immediately transferred to the place where the label: begins. This happens unconditionally.

Example programs:-

⇒ Write a program to check whether the given number is even or odd using goto statement.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int x;
    clrscr();
    printf("Enter x:");
    scanf("%d",&x);
    if(x%2==0)
      goto even;
    else
      goto odd;
    even: printf("Number is even");
          return;
    odd: printf("Number is odd");

}
```

Explanation:-

⇒ In the above program, if x is divisible by 2, then the control is transferred to the position 'even: and displays number is even.

Loop Control Statements:-

Loop:

A loop is defined as a block of statements which are repeatedly executed for certain number of times.

When to use Loops:-

⇒ If you want to perform the same operation repeatedly number of times, then loops will be used.

Steps in Loops:-

1. Loop variable: The variable used in the loop.
2. Initialization: It is the first step in which the starting con) initial value is assigned to the loop variable.
3. Test condition: Test condition dermines how many times the loop will execute.
4. Increment/Decrement: Updating the loop variable.

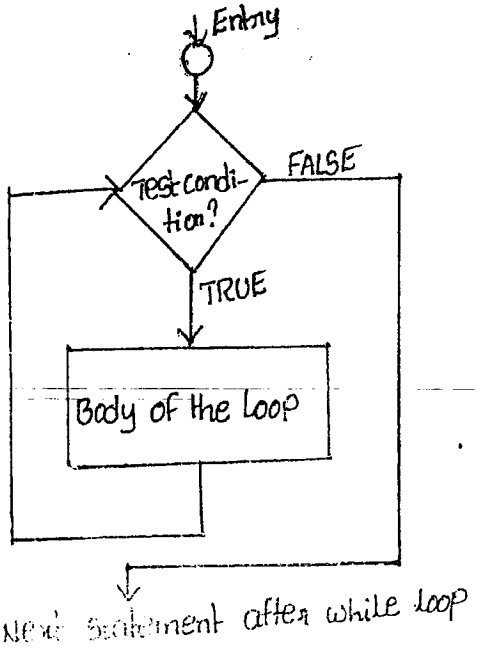
⇒ The C language supports 3 loop control statements:

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

While Loop:-

⇒ The basic format of while loop is:

```
while (test_condition)
{
    body of the loop
}
```



⇒ The while is an entry-controlled loop.

WORKING:-

⇒ The test-condition is evaluated first and if the condition is true, then the body of the loop is executed.

⇒ After execution of the body of the loop, the test-condition is once again evaluated and if it is true, the body of the loop is executed once again.

⇒ This process of repeated execution of the body continues till the test-condition becomes false.

⇒ If the test-condition becomes false, then the control is transferred out of the loop and the program execution continues with the statement immediately after the loop.

Example program:-

⇒ Write a program to display the message "welcome" 10 times.

```
/* Program to display the message "welcome" 10 times using while loop */
main()
{
   int i;
   i=1;
   while(i<=10)
   {
     printf("\n welcome");
     i=i+1;
   }
   getch();
}
```

OUTPUT:-

```
welcome
welcome
welcome
welcome
welcome
welcome
welcome
welcome
welcome
welcome
```

Explanation:-

In the above program, i is initialized to 1. The while loop checks the condition for i<=10. Since the condition is true, then the loop will be executed. ie. it displays the message "welcome" and increments i value by 1. Now i value becomes 2. Again it will check the condition, the condition is true and the body of the loop gets executed. This process continues till the condition becomes false.

② Write a program to find factorial of a given number using while loop.

```
/* Factorial of a given number */
main()
{
    int n, fact=1;
    clrscr();
    printf("Enter the number:");
    scanf("%d", &n);
    while(n>=1)
    {
        fact=fact*n;
        n--;
    }
    printf("\n Factorial of given number is: %d", fact);
    getch();
}
```

Sample Output:-

Enter the number: 4

Factorial of given number is: 24

Explanation:-

1st iteration:	2nd iteration	3rd iteration	4th iteration
n=4	n=3	n=2	n=1
Condition 4>=1 is true	3>=1 is true	2>=1 is true	1>=1 is true
∴ fact=1*4=4	∴ fact=4*3	∴ fact=12*2	∴ fact=24*1
n=n-1=3	n=3-1=2	=24	=24
		n=2-1=1	n=1-1=0

5th iteration:

n=0

0>=1 condition is false.

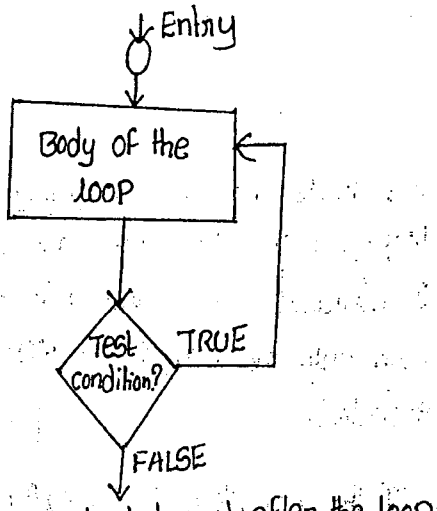
So control is transferred out of the loop.

⇒ Finally, it displays the value of fact i.e. 24

2. do-while loop:-

⇒ The basic format of do-while loop is:

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



WORKING OF do-while loop:-

⇒ In do-while loop, First the body of the loop is executed once. After that, it checks the condition at the end of the loop. If the condition is true, the body of the loop will be executed once again. This process continues as long as the condition is true.

When the condition becomes false, the loop will be terminated and the control goes to the statement immediately after the loop.

⇒ The do-while loop is an exit-controlled loop because test-condition is evaluated at the end of the loop.

Example Program:-

/* Program to display first 10 numbers using do-while loop */

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

OUTPUT:-

```
1
2
3
4
5
6
7
8
9
10
```

Difference between while and do-while:-

while	do-while
1. The while loop is an entry-controlled loop. i.e. in while loop the condition is evaluated first and if it is true then only the body of the loop will be executed.	1. The do-while loop is an exit-controlled loop. i.e. in do-while loop first the body of the loop is executed, after that at the end of loop the condition is evaluated.

2. If the condition is initially false, the while loop will not be executed atleast once.
Ex:
`i=10;`
`while(i<=10)`
`{`
`printf("welcome");`
`}`
→ Here, the while loop will not be executed since the condition is false
output: None
	2. The do-while loop will execute atleast once even if the condition is initially false.
	Ex:
	`i=10;`
	`do`
	`{`
	`printf("welcome");`
	`}while(i<=0)`
	→ Here, the do-while loop is executed one time even if the condition is false
	output: welcome

3. Syntax:
`while(condition)`
`{`
`statements;`
`}`
Note: while should not be end with semicolon.
	3. Syntax:
	`do`
	`{`
	`statements;`
	`}while(condition);`
	Note: The do-while loop should terminate with semicolon.

3. for Loop:

⇒ The for loop is another entry-controlled loop.

⇒ The general form of for loop is:

```
for( initialization ; test-condition ; increment/decrement)
{
        body of the loop;
}
```

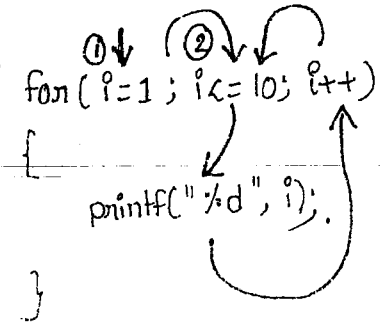
→ i.e updating loop variable.

WORKING:-

⇒ The execution of the for loop is as follows:

1. Initialization of the loop variable is done first using assignment statements such as i=10, count=0, j=0 etc. Initialization part is executed only once.
2. The test-condition is any relational expression such as i<10 that determines when to exit from the loop. The for loop continues to execute as long as the test condition is true. when the condition becomes false the loop is terminated and the execution continues with the statement after the loop.
3. When the body of the loop is executed, the control is transferred back to the for statement. The loop variable is updated and once again the test condition is checked. If the condition is true, the body of the loop is again executed. This process continues till the condition becomes false.

EX:



Example program:-

/* Program to display first 10 numbers in descending order */

```
main()
{
   int i;
   clrscr();
   for(i=10; i>=1; i--)
   {
      printf("\n %d", i);
   }
   getch();
}
```

OUTPUT:
10
9
8
7
6
5
4
3
2
1

Explanation:-

⇒ The value of i is initialized to 10 when the for loop execution starts.

⇒ Next the test condition i>=1 is evaluated. Initially the condition is true since i=10, 10>=1 is satisfied. So, the for loop will execute.

⇒ Upon executing the printf statement Compiler sends control back to the for loop where i value is decremented by 1. After decrementing the i value, the test condition is once again checked. If condition is true, the body is executed. This process continues till i value becomes 0.

Example program:-

/* Program to display all the even numbers from 1 to 10 */

```
main()
{  int i;
   for(i=2; i<=10; i=i+2)
   {
      printf("\n %d", i);
   }
}
```

OUTPUT:
2
4
6
8

JUMPS IN LOOPS:-

⇒ Loop performs a set of operations repeatedly until some condition becomes false.

⇒ Sometimes it is necessary to skip a part of the loop or exit the loop as soon as a certain condition occurs.

⇒ For example, consider you want to search for a particular name in a list of 100 names. The searching process should be terminated as soon as the desired name is found.

⇒ C supports the following jump statements:

1. break
2. continue
3. goto

1. break:-

⇒ The break statement is used to terminate the loop.

⇒ When a break statement is encountered inside a loop, the loop is immediately terminated and the program execution continues with the statement after the loop. i.e. the break statement skips from the loop in which it is defined.

⇒ The break statement is used with conditional if statement.

Ex:
```
while(condition)
{
------
------
if(condition)
    break;
------
------
}
------
```
Exit from loop

```
do
{
------
------
------
if(condition)
    break;
------
------
}while(condition);
------
```
Exit from loop

```
for(i=1;i<=10;i++)
{
------
------
------
if(condition)
    break;
------
------
}
------
```
Exit from loop

⇒ We can also use break statements inside nested for loops.

⇒ When break statement is used in nested loops, the control would terminate only from the loop in which it is defined.

Ex:

```
for(------)
{
  -------
  for(-----)
  {
    -----
    -----
    if(condition)
      break;
    ------
  }
  ------
  ------
}
```

Exit from inner most loop

```
while(condition)
{
  ------
  -----
  while(condition)
  {
    ------
    if(condition)
      break;
    ----
    ----
  }
  ------
  ------
}
```

Exit from inner loop

Example PROGRAMS:-

① Write a program to exit loop when you encounter 5.

```
main()
{
  int i;
  for(i=1; i<10; i++)
  {
    if(i==5)
      break;
    printf("%d", i);
  }
}
```

Output:
1
2
3
4

→ In the above program the loop will be terminated as soon as i becomes 5.

② Write a program to display the message 10 times using infinite loop.

```
main()
{
   int i=1;
   for( ; ;)
   {
      printf("\n Hello");
      if(i==10)
       break;
       i++;
   }
   getch();
}
```

→ In the above program the loop will be terminated as soon as i value becomes 10.

2. Continue statement:-

⇒ The continue statement is opposite to the break statement.

⇒ The continue statement is used to continue with the next iteration.

⇒ When continue statement is used inside a loop, it tells the compiler "skips the following statements and continue with the next iteration".

Ex:

```
while(condition)
{
   -------
   -------
   if(condition)
     continue;
   -------
   -------
}
```

continue with the next iteration

```
for(i=1; i<10; i++)
{
   -------
   -------
   -------
   if(condition)
     continue;
   -------
   -------
}
```

Example programs:-

⇒ Display all the numbers from 1 to n which are not divisible by 4.

```
main()
{
    int i,n;
    clrscr();
    printf("Enter n value:");
    scanf("%d",&n);
    for(i=1;i<=n;i++)
    {
        if(i%4==0)
          continue;
          printf("\n%d",i);
    }
    getch();
}
```

→ In the above program, if the number is divisible by 4 then it will not display that number, it will continue with the next iteration.

⇒ Write a program to display all the even numbers from 1 to 10 using continue statement.

```
main()
{
   int i;
   for(i=1;i<=10;i++)
   {
      if(i%2!=0)
         continue;
         printf("%d",i);
   }
}
```

Difference between break and continue:-

Break	Continue
1. Exits from current loop	1. Loop takes next iteration.
2. Control passes to next statement after the loop.	2. Control passes at the beginning of the loop.
3. Terminates the program.	3. Never terminates the program.

3. goto statement:-

⇒ We can also use goto statement to jump within a loop and to exit from the loop.

```
while(condition)
{
    ------
    ------
    if(error)
      goto stop;
    ------
    ------
    ------
    if(condition)
      goto abc;
    ------
    ------
    abc:
    ------
    ------
}
stop:
------
------
```

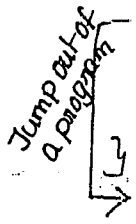
Jump within the loop

Exit from the loop.

JUMPING OUT OF THE PROGRAM

⇒ We can jump out of a program by using the library function exit().

Ex:
```
main()
{
    -----
    -----
    -----
    if(condition)
        exit(0);
    -----
    -----
}
```



⇒ The exit() function takes an integer value as its argument.

Normally 0 → indicates normal termination

non zero → indicates termination due to some error.

⇒ The use of exit() function requires the inclusion of header file <stdlib.h>.

Additional Features of for loops:-

The for loop has several additional features.

1. More than one variable can be initialized at a time in the for statement.

Ex:
```
for(i=10,j=0; i<10; i++)
{
    ----
}
```

→ Here the initialization section has two parts i=10 and j=0 which are seperated by comma.

2. Like in the initialization section, the increment/decrement section may also have more than one part.

Ex:
```
for(i=10,j=1; i<10; i--,j++)
{
    ----
}
```

3. Further more, the test condition may have any compound relation (→ more than one condition)

Ex:
```
for(i=10,j=1; i>=1 && j<=10; i--,j++)
{
    printf("\n %d \t %d", i,j);
}
```

Example program:-

```
/* Program to display numbers from 1 to 10 in ascending & descending order */
main()
{
    int i,j;
    clrscr();
    for(i=1,j=10; i<=10 && j>=1 ; i++,j--)
    {
        printf("\n %d \t %d", i,j);
    }
    getch();
}
```

OUTPUT:

1	10
2	9
3	8
4	7
5	6
6	5
7	4
8	3
9	2
10	1

4. Another Feature of for loop is that one or more sections can be omitted if necessary.

Ex:
```
i=1;
for( ; i<=10; i++)
{
  printf("\n %d", i);
}
```
⇒ Here, the initialization section is omitted in the for loop. The initialization has been done before the for loop.

Ex:
```
for(i=1; i<=10; )
{
  printf("\n %d", i);
  i=i+1;
}
```
⇒ Here, the increment section is omitted in the for loop. The increment section has been done inside the for loop.

Ex:
```
i=1;
for( ; i<=10 ; )
{
  printf("\n %d", i);
  i=i+1;
}
```
⇒ Here both initialization and increment sections are omitted in the for loop.

Note: Even if the sections are not present, the semicolons seperating the sections must remain.

5. Infinite loops:

⇒ If the test condition is not present, the for loop forms an infinite loop.

Ex:
```
for(i=1; ; i++)
{
  ------
}
```
⇒ Here there is no test condition. It leads to an infinite loop.

⇒ Ex: for (; ;) /* Infinite loop */

```
{
  ------
  ------
}
```

⇒
```
for(a=0; a<=10; )
{
   printf("%d", i);
}
```

→ This is also infinite loop, since a is neither increased nor decreased. So the condition is always true.

NESTED for LOOPS:-

⇒ We can also use loop within loops.

⇒ Nested for loops means, one for statement within another for statement.

Ex:

```
for(i=1; i<=10; i++)
{
  ------
  ------
  for(j=1; j<=5; j++)
  {
     -----
     -----
  }
  -----
  -----
}
```

inner loop

outer loop.

⇒ The number of iterations in this type of loops will be equal to the number of iterations in the outer loop multiplied by number of iterations in the inner for loop

⇒ In the above example, the outer loop will be executed 10 times and the inner loop will execute 5 times.

∴ Total number of iterations = 10x5 = 50 times.

⇒ The nesting may continue upto any desired level.

Example program:-

⇒ Write a program to display all the prime numbers between 1 to 100.

```
main()
{
  int i,j, count;
  clrscr();
  for(i=1; i<=100; i++)
  {
     count=0;
     for( j=1; j<=i; j++)
     {
        if(i%j==0)
        count++;
     }
     if(count==2)
     {
       printf("\n %d", i);
     }
  }
  getch();
}
```

⇒ Program to display the following pattern.

```
*
**
***
****
```

```
main()
{
  int i,j,n;
  clrscr();
  printf("Enter n value:");
  scanf("%d", &n);
  for(i=1; i<=n; i++)
  {
     for(j=1; j<=i; j++)
     {
       printf("*");
     }
     printf("\n");
  }
  getch();
```

ARRAYS

ARRAY:-

An array is a collection of homogeneous elements which shares the common name

(OR)

An array is a collection of similar data items.

⇒ All the elements in the array should be same data type.

SIGNIFICANCE:-

⇒ Ordinary variable can store only one value at a time. These variables can be useful to handle small amounts of data.

⇒ In many application, we need to handle large amounts of data.

For example consider we need to store marks of 60 students in a class. Using Ordinary variables we have to declare 60 variables to store marks of 60 students.

int m1, m2, m3, m4, ..., m60; → Because one variable can hold only one value at a time.

This increases program code. To avoid this it is better to use array.

By using arrays we can store marks of 60 students in a class under a single variable name.

Ex: int marks[60];

where marks is an array which stores marks of 60 students.

Examples where arrays can be used:

1. To store list of employees in an organization
2. List of temparatures recorded every hour in a day.
3. List of products
4. To store marks of students in a class.
5. List of customers and their phone numbers

⇒ Arrays can be used to represent a list of numbers or list of names.

ARRAY SUBSCRIPT:-

⇒ Subscript of an array is an integer expression (or) integer constant (or) integer variable that refers to the individual elements in the array.

Ex: int salary[100]; → subscript

→ The above array stores salaries of 100 employees in an organization.

→ We can access the individual salaries by writing index or subscript in brackets after the array name.

Ex: salary[5] → represents the salary of 5th employee

salary[30] represents the salary of 30th employee:

↳ subscript

CLASSIFICATION OF ARRAYS:-

⇒ We can use arrays to represent list of values and table of data in two, three or more dimensions.

⇒ Generally arrays are classified based on the dimensionality.

⇒ Dimensionality is determined by the number of subscripts present in an arr

⇒ Arrays are classified into: 1. One-dimensional arrays
2. Two-dimensional arrays
3. Multi-dimensional arrays.

⇒ If the array contains one subscript, then it is called one-dimensional arrays.
If the array contains two subscripts, it is called two-dimensional array.
Multi-dimensional arrays contains more than two subscripts.

ONE-DIMENSIONAL ARRAYS:-

⇒ One-dimensional arrays have only one subscript.

⇒ A list of items can be given one variable name using only one subscript and su

⇒ A list of items can be represented by a single variable name using only on subscript and such variables is called one-dimensional array (or) single-subs variable.

DECLARATION OF ONE-DIMENSIONAL ARRAYS:-

⇒ Arrays must be declared before they are used in the program.

Syntax: `data-type array-name[size];`

Where, Data-type → specifies type of elements that can be stored inside the arr such as int, float or char.

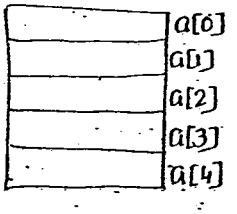
array-name → is a valid identifier

size → size indicates the maximum number of elements that can be stored inside the array.

Example: If we want to represent a set of 5 integer numbers by an array variable 'a', then we can declare the array as follows:

```
int a[5];
```

Now: the computer reserves five storage locations as shown below:



→ The values to the array elements can be assigned as follows:

a[0] = 10
a[1] = 20
a[2] = 30
a[3] = 40
a[4] = 50

⇒ The elements are stored in continuous memory locations.

a[0]	a[1]	a[2]	a[3]	a[4]
10	20	30	40	50
1000	1002	1004	1006	1008

⇒ The size of the array should be unsigned integer constant or a symbolic constant.

Ex: int number[10];
float height[50];
char name[10];

NOTE: Size should not be empty, it should not be negative value and floating-point value.

⇒ When we declare an array, the compiler allocates memory space based on the size and type of the array.

Total memory space allocated to an array = number of datatypes (bytes) occupied by the datatype * size of the array.

Ex: int a[10];

→ Total memory space allocated = 2*10 = 20 bytes.
(2 ↓ sizeof int, 10 ↓ no. of elements)

Ex: char name[10];

→ Total memory space allocated is: 1×10 = 10 bytes

Ex: float height[50];

→ Total memory space allocated = 4*50 = 200 bytes.

INITIALIZATION OF ONE-DIMENSIONAL ARRAYS:

⇒ After an array is declared, its elements must be initialized.

⇒ Assigning values to the array elements is known as initialization of arrays.

⇒ An array can be initialized at either of the following stages:

1. At compile time
2. At run time

1. Compile time initialization:-

⇒ We can initialize array elements at the place of their declaration itself.

Syntax: | Datatype array-name[size] = {list of values seperated by commas};

Example: int x[6] = {1, 2, 17, 14, 3, 8};
char name[5] = {'x', 'y', 'z', 'a', 'b'};

Accessing array elements:-

The individual elements of an array can be accessed by writing index or subscript in brackets after the array name.

Ex: int x[6] = {10, 100, 20, 30, 40, 50}

The subscript or index ranges from 0 to size-1.

i.e. x[0] = 10
x[1] = 100
x[2] = 20
x[3] = 30
x[4] = 40
x[5] = 50

Examples: int number[5] = {1, 2, 3, 4, 5};
Char C[4] = {'A', 'B', 'C', 'D'};
float height[3] = {5.1, 4.9, 6.3};

⇒ Suppose the list of elements specified are less than the size of the array, then that many elements are ~~assig~~ initialized. Remaining elements are assigned ~~garbage values~~ zero

Ex: int P[5] = {1, 2, 3};

P[0] = 1, P[1] = 2, P[2] = 3, P[3] = ~~garbage value~~ 0, P[4] = ~~garbage value~~ 0.

⇒ During compile time initialization, if we don't specify the size of the array, the compiler fixes the size of the array based on number of elements in the

Ex: int x[] = {1, 5, 7, 9};

Since the array x has 4 values in the list, the size of array is 4.

⇒ Any access to the array elements outside its boundaries (i.e. outside its limits) would not ~~access~~ cause any error. It results in unpredictable results.

Ex: int x[5] = {10, 20, 30, 40, 50};

Suppose if we try to access x[6] element, it will not show any error
↳ Here we are accessing beyond the limit

→ C performs no bounds checking. So you should ensure that the array indi are within the declared limits.

⇒ An array can be initialized at run time. This approach is used for initializing large arrays.

⇒ We can use scanf() function to initialize an array at runtime.

Ex: int x[3];

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

(OR)

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

⇒ Reading:

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

Displaying:

```
for(i=0; i<size; i++)
{
   printf("%d", a[i]);
}
```

Example program ①:-

⇒ Write a program to read and display array of 5 integers. Read array elements at compile time.

```
#include<stdio.h>
void main()
{
   int a[5] = {10, 20, 30, 40, 50}; /* Reading */
   clrscr();
   printf("%d %d %d %d %d", a[0], a[1], a[2], a[3], a[4]); /* Displaying */
   getch();
}
```

Example program ②

⇒ Write a program to read and display array of 5 real numbers (i.e. floating point value). Read array elements at run time.

```
#include<stdio.h>
void main()
{
   float a[5]; int i;
   printf("Enter 5 Real numbers:");
   for(i=0; i<5; i++)
   {
      scanf("%f", &a[i]);
   }
   printf("\n Array elements are:");
   for(i=0; i<5; i++)
      printf("\t %f", a[i]);
}
```

Example program ③

Write a program to display array elements in reverse order

```
#include<stdio.h>
#include<conio.h>
void main()
{
   int a[5],i;
   printf("Enter the array elements:");
   for(i=0;i<5;i++)
   {
     scanf("%d",&a[i]);
   }
   printf("Array elements in reverse order:");
   for(i=4;i>=0;i--) /*Display elements from size-1 to 0 i.e. from 4 to 1)
                                (5-1)
   {
     printf("\t%d",a[i]);
   }
   getch();
}
```

output:

```
Enter the array elements:
10 20 30 40 50
Array elements in reverse orde
50 40 30 20 10
```

Example ④

Write a program to implement a one-dimensional array and prints its index and corresponding value

```
#include<stdio.h>
void main()
{
   int a[5],i;
   printf("Enter array elements:");
   for(i=0;i<5;i++)
   {
     scanf("%d",&a[i]);
   }
   printf("Index\t Value");
   for(i=0;i<5;i++)
   {
     printf("\n%d\t%d",i,a[i]);   a[i] is value
                                  i is index
   }
   getch();
}
```

Input: Enter array elements: 13 12 25 17 9

output:

Index	value
0	13
1	12
2	25
3	17
4	9

⇒Write a program to calculate sum of array elements

```
#include<stdio.h>
Void main()
{
   int a[5], i, sum=0;
   printf("Enter array elements:");
   for(i=0;i<5;i++)
   scanf("%d",&a[i]);
   for(i=0;i<5;i++)
   {
      sum=sum+a[i];
   }
   printf("Sum of array elements is:%d", sum);
}
```

Input: Enter array elements:
1 2 3 4 5

Output:
Sum of array elements is: 15

Example⑥-

⇒Write a program to calculate average marks obtained by 50 students

```
#include<stdio.h>
Void main()
{
   int marks[50], i, sum=0;
   float average;
   printf("Enter 50 students marks");
   for(i=0;i<50;i++)
   {
      scanf("%d", &marks[i]);
   }
   for(i=0;i<50;i++)
   {
      sum=sum+marks[i];
   }
   average=sum/50;
   printf(" Average marks is:%f", average);
}
```

Example⑦-

⇒Write a program to calculate the total prices of 25 items which are stored in prices array.

```
#include <stdio.h>
Void main()
{
   float prices[25], total=0.0;
   int i;
   clrscr();
   printf("Enter prices of 25 items:");
   for(i=0;i<25;i++)
   {
      scanf("%f", &price[i]);
   }
   for(i=0;i<25;i++)
   {
      total=total+prices[i];
   }
   printf(" total price is: %f", total);
   getch();
}
```

Example(5):-

⇒ Write a program to find smallest and largest number in a list of integers.

```
#include<stdio.h>
#include<conio.h>
void main()
{
   int a[5], i, large, small;
   clrscr();
   printf("Enter array elements:");
   for(i=0;i<5;i++)
   {
     scanf("%d",&a[i]);
   }
   small=a[0];
   large=a[0];
   for(i=1;i<5;i++)
   {
      if(a[i]>large)
         large=a[i];
      if(a[i]<small)
         small=a[i];
   }
   printf("\n Smallest element in the array is:%d", small);
   printf("\n Largest element in the array is:%d", large);
   getch();
}
```

Input:-

Enter array elements: 100 5 105 107

output:

Smallest element in the array is: 5

Largest element in the array is: 107

Some more programs:

Try to practice the following programs:

① Write a program to calculate the sum of floating point numbers

② Write a program to display only -ve elements of an array.

③ Write a program to display only even numbers in an array.

④ Write a program to calculate sum of positive numbers and sum of negative nur in an array.

⑤ Write a program to add two integer arrays.

⑥ Write program to read and display a character array.

⇒ Write a program to sort the array elements in ascending order.

```
#include<stdio.h>
Void main()
{
   int a[10], n, i, temp, j;
   clrscr();
   printf("\n Enter number of array elements:");
   scanf("%d", &n);
   printf("\n Enter array elements:");
   for(i=0; i<n; i++)
   {
      scanf("%d", &a[i]);
   }
   for(i=0; i<n; i++)
   {
      for(j=i+1; j<n; j++)
      {
         if(a[i]>a[j])
         {
            temp=a[i];
            a[i] = a[j];
            a[j] =temp;
         }
      }
   }
   printf("\n Array elements in ascending order:\n");
   for(i=0; i<n; i++)
   {
      printf("\t %d", a[i]);
   }
   getch();
}
```

Input: Enter the number of array elements: 5

Enter array elements: 4 1 6 5 3

output: Array elements in ascending order: 1 3 4 5 6

⇒ Write a program to sort the array elements in descending order.

Logic:
```
for(i=0;i<n;i++)
{
  for(j=i+1;j<n;j++)
  {
    if(a[i]<a[j])
    {
      temp=a[i];
      a[i]=a[j];
      a[j]=temp;
    }
  }
}
```

⇒ Write a program to find k^{th} smallest element in an array

Logic:
1. Read array elements
2. Arrange the array elements in ascending order.
3. Print k^{th} element of sorted array.

⇒One-dimensional arrays can store list of values.

⇒There are situations, where we want to store a table of values. Two-dimensional arrays are used to represent or store table of values.

Two-dimensional array:

An array which contains two subscripts is known as two-dimensional array.

Example: Matrix

↳ Matrix contains two dimensions i.e. Rows & Columns.

$$A = \begin{array}{c} \\ \text{Row } 0 \\ \text{Row } 1 \end{array} \begin{array}{c} \begin{array}{cc} \text{Column } 0 & \text{Column } 1 \end{array} \\ \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \end{array}$$

In matrix, a particular element is represented by two subscripts. For example A_{ij} refers to the value in the i^{th} Row and j^{th} column in matrix A.

$A_{00} = 1$, $A_{01} = 2$, $A_{10} = 3$, $A_{11} = 4$

⇒In C language matrix can be represented by two-dimensional arrays.

DECLARATION OF TWO-DIMENSIONAL ARRAYS:-

⇒Two-dimensional arrays are declared as follows:-

Syntax: `datatype array-name[rowsize][columnsize];`

Ex: `int a[3][3];` → Contains 3 rows and 3 columns
`int b[3][2];` → Contains 3 rows and 2 columns
`char c[5][4];` → Contains 5 rows and 4 columns.

⇒Total number of elements in a two-dimensional array is = Rowsize * Columnsize

Space occupied by the two-dimensional array = number of bytes occupied by the datatype * total number of elements.

Example: `int a[5][4];`
total number of elements = 5*4 = 20
Space = 2*20 = 40 bytes.

⇒Each dimension of the array is indexed from 0 to its maximum-size − 1.

⇒The first index selects the row and second index selects column within that row.

INITIALIZATION:-

1. Compile time initialization:-

⇒ Initializing two-dimensional arrays at the time of declaration itself.

Ex: int table[2][3] = {0,0,0,1,1,1};

The above statement initializes first row to 0 and second row to 1. The initi
ion is done row by row. i.e. $\begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$

⇒ We can also initialize two-dimensional array in the form of matrix as shown bel

Ex: int table[2][3] = { {0,0,0}, {1,1,1} };

int a[3][2] = { {4,5}, {1,7}, {8,9} };

⇒ When the array is completely initialized with all values, then no need to specify size of the first dimension.

Ex: int a[][3] = { {4,5,6}, {10,17,1} };

⇒ If the values are missing in initialization, they are automatically set to 0.

Ex: int a[2][3] = { {1,2}, {4} };

a[0][0] = 1 a[1][0] = 4
a[0][1] = 2 a[1][1] = 0
a[0][2] = 0 a[1][2] = 0

Accessing two-dimensional arrays:-

⇒ Individual elements of two-dimensional arrays can be accessed by using two dimensions.

Ex: a[i][j] represents elements in i^{th} row and j^{th} column.

array name ← a[ ][ ] → Row, Column

Ex: int a[2][2] = { {10,20}, {30,40} }

(00 = 10, 01 = 20, 10 = 30, 11 = 40)

	0	1
0	10	20
1	30	40

i.e. a[0][0] = 10
a[0][1] = 20
a[1][0] = 30
a[1][1] = 40

Arrangement of two-dimensional array elements in memory:-

⇒ Either it is a two-dimensional array or one-dimensional array, all the array elements are stored in continuous memory locations.

Ex: int a[2][2] = { {10,20}, {30,40} };

a[0][0]	a[0][1]	a[1][0]	a[1][1]
10	20	30	40

3. Run-time initialization:-

⇒ Two-dimensional arrays can be initialized at run time using scanf function.

Reading:-

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

Displaying:-

```
for(i=0;i<rowsize;i++)
{
   for(j=0;j<columnsize;j++)
   {
      printf("%3d",a[i][j]);
   }
   printf("\n");
}
```

Example program:-

⇒ Write a program to read and display 3x3 matrix.

```
#include<stdio.h>
#include<conio.h>
Void main()
{
   int a[3][3],i,j;
   clrscr();
   printf("Enter a 3x3 matrix:");
   for(i=0;i<3;i++)
   {
      for(j=0;j<3;j++)
      {
         scanf("%d",&a[i][j]);
      }
   }
   printf("The entered matrix is:\n");
   for(i=0;i<3;i++)
   {
      for(j=0;j<3;j++)
      {
         printf("%3d",a[i][j]);
      }
      printf("\n");
   }
   getch();
}
```

Reading

displaying

Input:-

Enter a 3x3 matrix: 1 4 5
7 8 9
10 2 3

Output:

The entered matrix is:

1 4 5
7 8 9
10 2 3

⇒ Example program:-

⇒ Write a program to perform addition of two matrices.

```
#include<stdio.h>
Void main()
{
   int a[10][10], b[10][10], c[10][10], r1, r2, c1, c2, i, j;
   clrscr();
   printf("Enter number of rows & columns in matrix a:");
   scanf("%d %d", &r1, &c1);
   printf("\n Enter matrix a:");
   for(i=0; i<r1; i++)
   {
      for(j=0; j<c1; j++)
      {
         scanf("%d", &a[i][j]);
      }
   }
   printf("\n Enter number of rows and columns in matrix b:");
   scanf("%d %d", &r2, &c2);
   printf("\n Enter matrix b:");
   for(i=0; i<r2; i++)
   {
      for(j=0; j<c2; j++)
      {
         scanf("%d", &b[i][j]);
      }
   }
   if(r1==r2 && c1==c2)
   {
      printf("matrix addition is possible");
      for(i=0; i<r1; i++)
      {
         for(j=0; j<c1; j++)
         {
            c[i][j]=a[i][j]+b[i][j];
         }
      }
      printf("\n Addition of two matrices:\n");
      for(i=0; i<r1; i++)
      {
         for(j=0; j<c1; j++)
         {
            printf("%3d", c[i][j]);
         }
         printf("\n");
      }
   }
   else
   {
      printf("matrix addition is not possible");
   }
   getch();
}
```

```
#include<stdio.h>
void main()
{
 int a[10][10], b[10][10], c[10][10], i, j, k;
 int r1, r2, c1, c2;
 clrscr();
 printf("Enter no.of rows & columns in a:");
 scanf("%d %d", &r1, &c1);
 printf("\n Enter matrix a:");
 for(i=0; i<r1; i++)
 {
   for(j=0; j<c1; j++)
   {
     scanf("%d", &a[i][j]);
   }
 }
printf("\n Enter no.of rows & columns in b:");
 scanf("%d %d", &r2, &c2);
printf ("\n Enter matrix b:");
 for(i=0; i<r2; i++)
 {
   for(j=0; j<c2; j++)
   {
     scanf("%d", &b[i][j]);
   }
 }
if( c1==r2)
{
   printf("matrix multiplication is possible");
   for(i=0; i<r1; i++)
   {
      for(j=0; j<c2; j++)
      {
         c[i][j]=0;
         for(k=0; k<c1; k++)
           c[i][j]=c[i][j]+a[i][k]*b[k][j];
      }
   }
   printf("\n Resultant matrix is:\n");
   for(i=0; i<r1; i++)
   {
      for(j=0; j<c1; j++)
      {
        printf("%4d", c[i][j]);
      }
      printf("\n");
   }
}
else
{
 printf("matrix multiplication is not possible.
}
getch();
}
```

⇒ Write a program to generate transpose of a matrix.

```
#include<stdio.h>
Void main()
{
  int a[3][3], b[3][3], i, j;
  clrscr();
  Printf("Enter a matrix:");
  for(i=0; i<3; i++)
  {
    for(j=0; j<3; j++)
     scanf("%d", &a[i][j]);
  }
  for(i=0; i<3; i++)
  {
     for(j=0; j<3; j++)
     {
        b[i][j] = a[j][i];
     }
  }
  Printf("Transpose of given matrix is:\n");
  for(i=0; i<3; i++)
  {
     for(j=0; j<3; j++)
     {
        printf("%d", b[i][j]);
     }
     printf("\n");
  }
  getch();
}
```

Input:

Enter a matrix: 1 2 3
4 5 6
7 8 9

Output:

Transpose of given matrix is:

1 4 7
2 5 8
3 6 9

HOME WORK:-

① Write a program to perform subtraction of two matrices.

② Write a program to calculate sum of all elements in a matrix.

③ Write a program to display diagnol elements and its sum in a matrix.

↳ if(i==j)
printf("%d", a[i][j]);

a[1][1]
a[2][2]
a[3][3]

④ Write a program to find whether the given matrix is symmetric or not

$A = A^T$ ⇒ symmetric

MULTI-DIMENSIONAL ARRAYS:-

⇒ An array which contains more than two subscripts is known as multi-dimension array.

⇒ Multi-dimensional arrays are used to represent survey data.

Declaration:-

Multi-dimensional arrays are declared as follows:

Syntax: datatype array-name [S1][S2][S3].....[Sm];

→ where Si is the size of i^{th} dimension.

Examples:- int survey[3][5][12];

float table[5][4][3][4];

Here survey is a 3-dimensional array, which contains 180 integer elements and table is a 4-dimensional array.

Example①:-

To represent a survey data of rainfall during the last 3 years in 5 cities from January to december, the array can be declared as follows:

```
int survey[3][5][12];
        ↓  ↓  ↓
      year city month
```

year1

month / City	1	2	3	...	12
1					
2					
3					
4					
5					

year2

month / City	1	2	3	...	12
1					
2					
3					
4					
5					

year3

month / City	1	2	...	12
1				
2				
3				
4				
5				

For example, the element survey[2][3][10] represents the rainfall in the month of october during the second year in city 3.

Example②:-

To represent gold rates during the last 2 years for from January to december every day, the array can be declared as follows:

int gold_rate[2][12][31];

INITIALIZATION:-

1. Compile time initialization:-

Multi-dimensional arrays can be initialized at compile time

Ex: int a[3][3][3] = { {{1,2,3}, {4,5,6}, {7,8,9}},
{{1,1,1}, {1,7,9}, {10,6,5}},
{{4,2,8}, {1,5,11}, {3,6,9}}
};

2. Run-time initialization:-

Multi-dimensional arrays can be initialized at run time using scanf function

Ex:
```
int x[3][2][5];
for(i=0; i<3; i++)
{
   for(j=0; j<2; j++)
   {
      for(k=0; k<5; k++)
      {
         scanf("%d", &x[i][j][k]);
      }
   }
}
```

FUNCTIONS UNIT - 3

→ Library Functions
→ Need for user-defined functions
→ A multi-function program
→ Elements of user-defined functions
→ Definition of functions
→ Return values and their types
→ Function calls
→ Function declaration
→ Category of functions
→ Nesting of functions
→ Recursion
→ Passing arrays to functions
→ The Scope, visibility and Lifetime of variables
→ Multifile programs
→ Prepnocesson commands

Functions:-

A function is a self-contained block of one or more statements that performs a particular task.

Example : scanf() → scanf() function reads the input
clrscr() function is used to clear the screen.

⇒ A 'C' program is a collection of functions.

⇒ Basically functions are divided into 2 types:

1. Library functions:

Library functions are predefined functions which are defined in C library. Examples: printf()
scanf()
clrscr()
pow()
sqrt() etc

→ Library functions are developed by the developers.
→ As a programmer we can't modify the library functions.
→ All the library functions are pre-defined in the headerfiles. If you are using any library functions, include the corresponding header file using "#include".

2. User-defined functions:-

The functions which are defined by the user or programmer are known as user-defined functions.

Example : main()

LIBRARY FUNCTIONS:

⇒ Library functions are pre-defined functions.

⇒ The main advantage is: code Reuse → Reusing the functions which are already been written and tested.

⇒ C allows reuse by providing many predefined functions that can be used to perform mathematical computations.

⇒ Some of the library functions are:

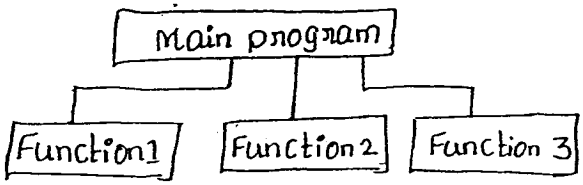
Function	Standard header file	Purpose	Example	Argu ty
abs(x)	<stdlib.h>	Returns the absolute value of its integer argument	If x = -5 abs(x) = 5	int
ceil(x)	<math.h>	Returns the smallest integral value that is not less than x	If x = 45.23 ceil(x) = 46.00	dou
floor(x)	<math.h>	Returns the largest integral value that is not greater than x.	If x = 45.23 floor(x) = 45.00	doul
sqrt(x)	<math.h>	Returns the non-negative square root of x (i.e. $\sqrt{x}$) for x ≥ 0	Ex: If x = 4.0 sqrt(x) = 2.0	doul
pow(x,y)	<math.h>	Return x^y	If x = 2, y = 3 pow (x,y) = 8	doub d
fabs(x)	<math.h>	Returns the absolute value of its type double argument	If x = -8.432 fabs(x) = 8.432	doul
exp(x)	<math.h>	Returns e^x, where e = 2.71828	If x = 1 exp(x) = 2.71828	dou
log(x)	<math.h>	Returns the natural logarithm of x for x > 0.	If x = 2.71828 log(x) = 1.0	dou
log10(x)	<math.h>	Returns the base-10 logarithm of x for x > 0	If x = 100.0 log10(x) = 2.0	dou
sin(x)	<math.h>	Returns the sine of angle x.	Note: The angle must expressed in radian	
cos(x)	<math.h>	Returns the cosine of angle x.		
tan(x)	<math.h>	Returns the tangent of angle x		

⇒ We already know that every C program must have a main function to indicate where the program has to begin its execution. It is possible to write any program using only main functio. But it leads to number of problems

- The program may become too large and complex
- The task of testing, debugging and maintaining becomes difficult

To avoid this, using functions large programs can be divided into subprograms. These subprograms are coded independently and later combine into a single unit. It is easy to understand, debug and test the subprograms

⇒ There are situations, where certain type of operations or calculation are repeated



⇒ There are situations, where certain type of operations or calculations are repeated at many points throughout a program.

For example consider, in a program we might use factorial of a numbe at several points in the program. we may repeat the program statements whenever they are needed. It increases program code.

To reduce the program code write the particular block of statements in a user-defined function and call the function whenever required. This reduces both program size and time.

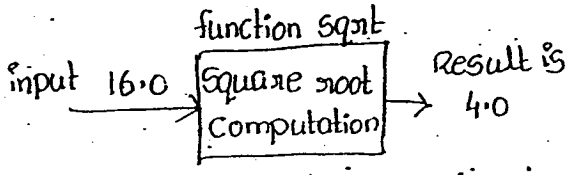
⇒ The main advantage is code Reusability. i.e. A function may be used by many other programs and any number of time.

⇒ It is easy to locate and isolate the faulty functions.

A MULTI-FUNCTION PROGRAM:

⇒ Once a function has been designed and packed, it can be treated as a 'black-box'.

Function takes some data from the main program and returns a value. The inner details of function are invisible to the rest of the program



All that the program knows about a function is: what goes in & what comes out.

⇒ Every C program is a collection of one or more functions.

⇒ For example, consider the following function printline():

```
void printline()
{
   int i;
   for(i=0; i<20; i++)
   printf("-");
   printf("\n");
}
```

The above function print a line of 20 character length. This function can be used in a program as follows:

```
main()
{
   void printline(); /* Function declaration */
   clrscr();
   printline(); /* Function call */
   printf("This shows the use of C functions\n");
   printline(); /* Function call */
   getch();
}
```

This program will print the following output:

```
--------------------
This shows the use of C functions
--------------------
```

The above program contains 2 user-defined functions: main() function
printline() function

We know that the program execution always begins with the main() functio During execution of the main, it calls printline function, which indicates that the function printline is to be executed. At this point, the working of callin function is stopped and the program control is transferred to the calle function printline. After executing the printline function, the control is transferred back to the main and continues the execution. After execut the printf() function, again it calls printline(). Now the control is again transferred to the printline function for printing the line.

⇒ Any function can call any other function. A 'called function' can also call another function.

```
Example: void main()
         {
           void message();
           void message1();
           clrscr();
           message();
           getch();
         }
         void message()
         {
           printf("Hai");
           message1();
         }
         void message1()
         {
           printf("\n Hello");
         }
```

① call → void message()

② call → void message1()

③ Return

Sample output:

```
Hai
Hello
```

In the above program, main function calls message(), which calls message1 i.e. a called function can also call another function.

⇒ A function can be called more than once.

```
Example: main()
         {
           void message();
           clrscr();
           message(); /* Function call */
           message(); /* Function call */
           message(); /* Function call */
           getch();
         }
         void message()
         {
           printf("\n This is a message");
         }
```

Output : This is a message
This is a message
This is a message

⇒ A function can call itself.

Ex:
```
main()
{
  printf("Hai");
  main();   → Here the main function calls itself.
}
```

Important Note:

→ A function can be called from any other function, but a function can't be defined inside a another function.

Ex:
```
main()
{
  printf("Hai");
void message()
{
  printf("This is a message");
}
}
```
} You should n't define a function inside another function.

→ The order in which the functions are defined in a program and the order in which they get called need not be the same.

Ex:
```
main()
{
  message1();
  message2();
}
message2()
{
  printf("\n This is message2");
}
message1()
{
  printf("\n This is message1");
}
```

ELEMENTS OF USER-DEFINED FUNCTIONS:

⇒ Functions are classified as one of the derived data types in C.

⇒ Like variables functions should declared and defined before using them in a program.

Similarities between variables and functions in C:

1. Both function names and variable names must be valid identifiers. so they must follow the rules.
2. Like variables, functions also have types(such as int, char) associated with them
3. Like variables, function names and their types must be declared and defined before they are used in a program.

How to use User-defined functions:-

To make use of user-defined functions, we need to establish 3 elements related to functions: 1. Function definition.
2. Function call.
3. Function declaration

FUNCTION DEFINITION:-

Function definition is also known as function implementation. Function definition consists of set of statements that are specially written to implement a particular task.

A general format of function definition is:

Syntax:

```
function-type function_name(parameter list)   => Function header
{
    local variable declaration;
    executable statement1;
    executable statement2;
    ----------
    ----------
    return statement;
}
```

=> Function Body

Function Header:

The function header consists of 3 parts:

1. Function type (also known as return type)
2. Function name
3. Parameter list

1. Function type:

The function type specifies the type of value returned by the function (like int or float or double).

If the function is not returning anything, then we need to specify the return type as void.

If the return type is not explicitly specified, C assumes that by default all functions returns an integer value.

The value returned is the output produced by the function.

2. Function name:

The function name is a valid C identifier. Therefore while giving function names you should follow the rules. The function name should be related to the task performed by the function.

3. Parameter list:

The parameter list declares the variables that will receive the data sent by the calling program.

Parameters are also known as arguments. Parameters serve as input data to the function to carry out the specified task.

The parameter list contains declaration of variables seperated by con

Example: ① `int sum(int a, int b)`
`{`
`------`
`------`
`}`

The above function sum takes two integer values as input and returns the result.

② `float quadratic(int a, int b, int c)`
`{`
`------`
`------`
`}`

⇒ A function need not always receive values from the calling program. In such cases, the parameter list will be empty. To indicate that the parameter list is emptly, we use void keyword.

Ex: `void sum(void)` (or) `void sum()` → This value doesn't return any value

`{`
`------`
`}`

→ Empty. It doesn't take any input.

FUNCTION BODY:-

The function body contains the declarations and statements necessary for performing the required task.

The function body contains:

1. Local variable declarations:
   A local variable is a variable that is declared inside a function.
2. Executable statements:
   - Set of statements that performs the task of the function.
3. Return statement:
   A return statement that returns the value.

Example: ①
```
void display(void)
{
    printf("No type, no paramenters");   /* No local variables */
                                         /* No return statement
}
```

②
```
int sum(int a, int b)
{
    int c;  /* Local variable declaration */
    c = a+b; /* Executable statements */
    return c; /* Returns the result */
}
```

RETURN VALUES AND THEIR TYPES

A function may or may not send back any value to the calling function. If a function send any value, it done through the return statement.

The return statement can take one of the following forms.

1. return;
   The above statement doesn't return any value, it acts as the closing brace of the function.
2. return(expression);
   The above statement returns the value of the expression.

Ex:
```
int sum(int a, int b)
{
    int c;
    c = a+b;
    return c;
}
```
The above function returns the value of c

(OR)
```
int sum(int a, int b)
{
    return(a+b);
}
```
↳ This is also valid.

A function may have more than one return statements. This situation arises when the value returned is based on certain condition.

Example:
```
if(a>b)
    return a;
else
    return b;
```

⇒ When a function reaches its return statement, the control is transferred back to the calling function.

FUNCTION CALL:-

⇒ To use any user-defined function, we need to call the function at a required place in the program.

⇒ A function can be called by simply using the function name followed by a list of parameters if any, enclosed in parenthesis.

Example:
```
sum(2,3);
sum();
sum(a,b);
sum(10,b);
sum(10+5,12);
sum(sum(2,3),6);
```

Note: The parameters used in the function call may be values, variables or expressions.

⇒ When the compiler encounters a function call, the control is transferred to the called function. The called function is then executed line by line.

FUNCTION DECLARATION:-

⇒ Like variables, all functions in a C program must be declared before they are invoked.

⇒ Function declaration is also known as function prototype.

Syntax: function_type functionname(parameter list);

⇒ A function prototype tells the C compiler the following things:

1. The type of value returned by the function.
2. The function name
3. Information about the parameters or arguments.

Ex:
```
void addition(int x, int y); /*Function prototype*/
void mul(int m, int n);
int largest(int a, int b);
float Average(int a, int b, int c);
```

The following example shows the working of functions.

```
void main()
{
    int sum(int, int);  /* Function declaration */
    int x;
    x = sum(10,5);      /* Function call */
    printf("In Sum is: %d", x);
    getch();
}
int sum(int a, int b)   /* Function definition */
{
    int c;
    c = a+b;
    return c;
}
```


Explanation:-

Calling function:-

The function which is calling another function is known as calling function.

Called function:

The function which is invoked (i.e. called) by the calling function is known as called function.

In the above example main() is the calling function
sum() is the called function

Formal arguments (or) Formal parameters:-

The parameters used in function declaration and function definition are called formal parameters (or) formal arguments.

Actual arguments (or) Actual parameters:

The parameters used in function call are called actual parameters (or) actual arguments.

⇒ The formal and actual parameters must match exactly in type, order and number. Their names need not be same.

⇒ The values of actual arguments are assigned to the formal arguments on a one to one basis, starting with the first argument.

WORKING OF FUNCTIONS:-

We know that program execution always starts with a main() function During the execution of a program, whenever a function is called the working of the calling function is stopped and the control is transferred to the called function.

The called function is then executed line by line. When the execution c the called function is completed or when it reaches return statement, the control is transferred back to the calling function and continues the executi

The values of the actual arguments passed by the calling function are received by the formal arguments of the called function. The called function operates on formal arguments.

NOTE:-

1. If the actual arguments are more than the formal arguments, then the extra actual arguments are discarded.
2. If the actual arguments are less than the formal arguments, then the unmatched formal arguments are initialized to some garbage values.
3. Any mismatch in data type may also result in passing of garbage values.
   → No error message will be generated.
4. The formal arguments must be valid variable names. The actual argume may be variable names, expressions or constants.
5. The variables used in actual arguments must be assigned values befor they are passed.

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:

1. Functions with no arguments and no return values
2. Functions with arguments and no return value
3. Functions with arguments and return value
4. Functions with no arguments but return value
5. Functions that return multiple values.

1. FUNCTIONS WITH NO ARGUMENTS AND NO RETURN VALUE:-

⇒ When a function has no arguments, it doesn't receive any data from the calling function.

⇒ Similarly, when the called function doesn't return any value, the calling function doesn't receive any data from the called function.

⇒ There is no data transfer between the calling function and the called function.

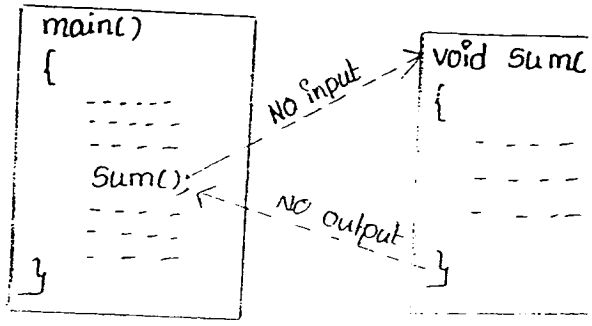


Figure: No data communication between functions

Example:- Program to find Sum of two numbers using functions with no arguments and no return value.

```
#include<stdio.h>
Void main()
{
    Void sum();
    clrscr();
    sum();
    getch();
}
Void sum()
{
    int a, b, c;
    Printf("Enter a, b values:");
    Scanf("%d %d", &a, &b);
    c=a+b;
    Printf("Sum is: %d", c);
}
```

Explanation:

→ In this program, the function sum() has no arguments. So it doesn't receive any data from the main function.

→ The function sum() doesn't return any value to the main() function. So return type is specified as void.

```
Void sum();
     └→ No arguments
└→ No return value
```

3. FUNCTIONS WITH ARGUMENTS BUT NO RETURN VALUE:-

⇒ In this case, the data are transferred from calling function to the called function.

⇒ The called function receives some data from the calling function. But it doesn't return any value back to the calling function.

Ex: void sum(int x, int y) → Receives the data (or) input from calling func
↓
Doesn't return any value.

⇒ The nature of communication between the calling function and the called function is as follows:

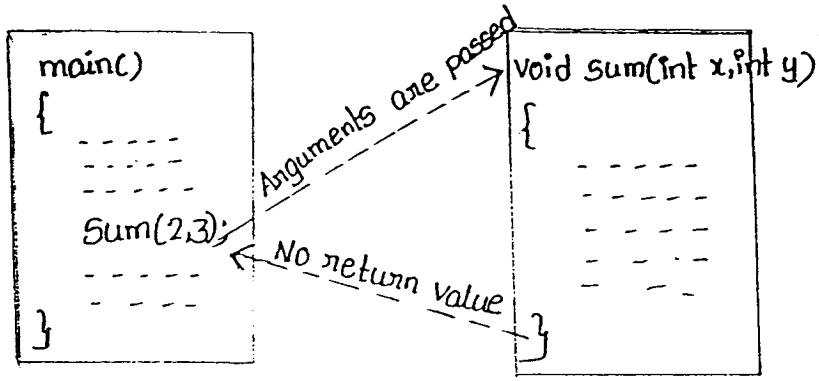


Figure: One-way data communication

Example:

```
/* Program to find the sum of two integers using function with arguments
   and no return value */
#include<stdio.h>
Void sum(int a, int b);
Void main()
{
   int x,y;
   clrscr();
   printf("Enter x,y values:");
   scanf("%d %d",&x,&y);
   sum(x,y);
   getch();
}
Void sum(int a, int b)
{
   int c;
   c=a+b;
   printf("Sum is: %d", c);
}
```

Explanation:-

→ In this program the function sum() receives 2 x,y values as input frc the main() function.

→ But the main() function doesn't receive any value from the sum();

FUNCTIONS WITH ARGUMENTS AND WITH RETURN VALUES:-

⇒ In this case, the data is transferred between the calling function and the called function.

⇒ The called function receives some data from the calling function and returns a value back to the calling function. There is a two-way data communication between the functions.

Ex: The function definition like as follows:

```
int sum(int x, int y)
```
→ Receives data from the calling function

↓

Returns a value back to the calling function

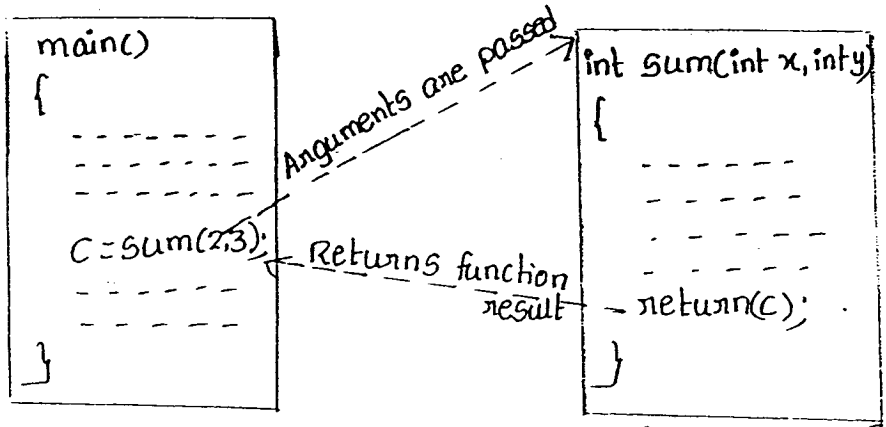


Figure:- TWO-WAY DATA COMMUNICATION BETWEEN FUNCTIONS

Example:

/* Program to calculate sum of two integers using functions with arguments and with return value */

```
#include<stdio.h>
Void main()
{
  int sum(int a,int b);
  int x, y, z;
  clrscr();
  printf("Enter x,y values:");
  scanf("%d %d",&x, &y);
  z = sum(x,y);
  printf("Sum is: %d", z);
  getch();
}
int sum(int a, int b)
{
    int c;
    c=a+b;
    return c;
}
```

Explanation:-

⇒ In this program, the function sum() receives two integer values as input from the main() function.

⇒ The function sum() calculates the sum of two integer values and returns result back to the main() function.

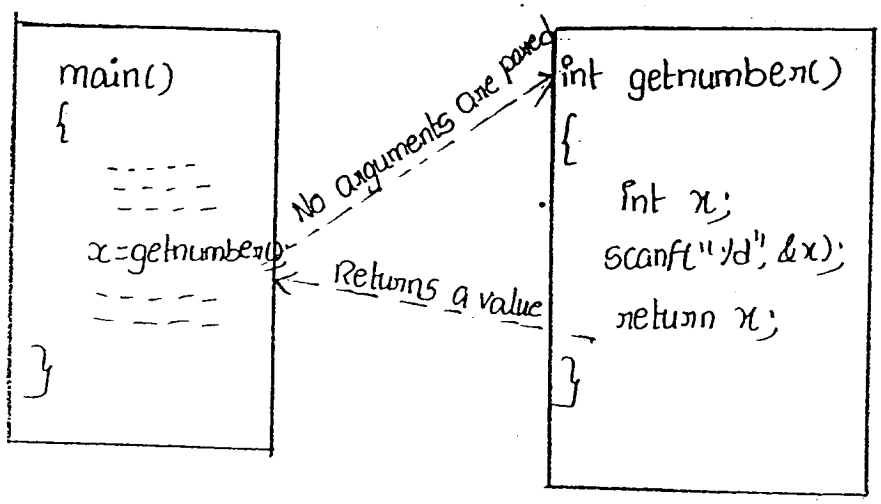
⇒ The main() function receives the returned value and that value is assigned to variable 'z'.

4. FUNCTIONS WITH NO ARGUMENTS BUT RETURN A VALUE:-

⇒ In this case, the called function doesn't take any arguments from the calling function, but returns a value to the calling function.

Ex: getchar() → getchar() function has no arguments but it returns a character.

Ex: int getnumber();
↓ → The function has no arguments
Function returns an integer value to the calling function.



Example program:-

```
#include<stdio.h>
void main()
{
   int getnumber();
   int a;
   a=getnumber();
   printf("a=%d", a);
   getch();
}
int getnumber()
{
    int x;
    printf("Enter a value:");
    scanf("%d", &x);
    return x;
}
```

NESTING OF FUNCTIONS:-

⇒ C allows nesting of functions.
⇒ Nesting of functions means calling one function inside another function

```
main()
{
    --------
    --------
    function1();
    --------
    --------
}
function2()
{
    --------
    --------
    function2();
    --------
    --------
}
function2()
{
    --------
    --------
    --------
}
```

⇒ Nesting of functions. function1() calls function2().

Figure: Nesting of functions

As shown in the above figure main function calls function1, which calls function2. There is no limit. Any function can call any another function.

Example:

Program to calculate the Ratio of $\frac{a}{b-c}$. The ratio cannot be evaluated if (b-c)=0.

```
float ratio(int x, int y, int z);
int difference(int x, int y);
void main()
{
    int a,b,c;
    clrscr();
    printf("Enter a,b,c values:");
    scanf("%d %d %d", &a, &b, &c);
    printf("Ratio is: %f", ratio(a,b,c));
}
```

```
float ratio(int x, int y, int z)
{
    if (difference(y,z)!=0)
        return (x/(y-z));
    else
        return (0.0);
}
int difference(int y, int z)
{
    if((y-z)!=0)
        return (1);
    else
        return (0);
}
```

Explanation:

The above program contains 3 functions : main()
ratio()
difference().

→ main() reads the values of a,b,c and calls the function ratio to calculate the value a/(b-c).

→ The ratio can't be evaluated if (b-c)=0. So ratio calls another function difference to test whether the difference (b-c) is zero or not.

→ The function difference return 1 if the difference (b-c) is not equal to zero, otherwise return zero to the function ratio.

→ The ratio calculates the ratio a/(b-c) value and returns the result if it receives 1. If the function receives '0' it returns 0.0 to main function.

Nesting of function calls:-

Nesting of function calls is also possible.

Example: `x = sum(sum(2,3),6);`

The above statement represents two sequential function calls. The inner function call is evaluated first (i.e. first it evaluates sum(2,3)) and the returned value is again used as actual argument in the outer function call. (ie. x=sum(5,6) = 11).

RECURSION:-

⇒A function that calls itself is known as recursive function and the process of calling function itself is known as recursion.

Ex: #include<stdio.h>
```
void main()
{
   printf("This is an example of recursion");
   main();
}
```

In the above program, main() function calls itself repeatedly infinite number of times. The user has to stop the execution abnormally (or) there must be some condition for stopping the execution -

⇒A useful example of recursion is the evaluation of factorial of a number.

Program:-

```
/* Factorial of a given number using recursion */
#include<stdio.h>
void main()
{
   int n;
   int factorial(int);
   clrscr();
   printf("Enter a number:");
   scanf("%d",&n);
   printf("The factorial of a given number is: %d", factorial(n));
   getch();
}
int factorial(int x)
{
   int fact;
   if(x==1)
      return 1;
   else
      fact = x*factorial(x-1);
   return fact;
}
```
→ /* Recursion */

Let us assume n=3.

Since n≠1 the statement fact=3*factorial(3-1) will be executed.

Again it includes a function call with n=2. This call will return 2*factoria

Once again the factorial is called with n=1. This time the function returns

⇒ The sequence of operations are:

fact = 3 * factorial(2)

= 3 * 2 * factorial(1)

= 3*2*1

= 6.

Advantages:-

1. Extremely useful when applying the same solution to the subsets of the problem.

~~3. Length~~

Disadvantages:-

1. Recursive functions can create infinite loops.
2. It requires extra storage space.
3. Recursive functions can create stack overflow.
4. If the programmer forgets to specify the exit condition in the recursive function, then the program will execute infinite number of time
5. It is difficult to trace the logic of the program.
6. It is difficult to debug the code containing recursion.
7. Often confusing.

PASSING ARRAYS TO FUNCTIONS:-

Like the values of simple variables, it is also possible to pass the elements of an array to a function.

Passing One-dimensional Arrays to functions:-

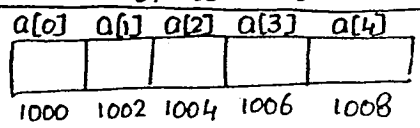
To pass a one-dimensional array to a called function, it is sufficient to pass the array name without any subscripts, and the size of the array as arguments.

Example: `int a[5] = {1, 2, 3, 4, 5};`

`largest(a,5);` → will pass the whole array a to the largest function
- 5 → size of the array
- a → Name of the array

⇒ In C, the name of the array represents the address of its first element.

Ex: `int a[5];`



a = &a[0] = 1000
a+1 = &a[1] = 1002
a+2 = &a[2] = 1004
a+3 = &a[3] = 1006
a+4 = &a[4] = 1008

→ By passing the array name, we are passing the address of the array to the called function. Any changes in the array in the called function will reflect the original array.

RULES:-

1. The function must be called by passing only the name of the array and its size.

   Ex: `largest(a, n);`

2. The function declaration must show that the argument is an array.

   Ex: `int largest(int a[], int n);`

3. In the function definition, the formal parameter must be an array type. The size of the array need not be specified in the square brackets.

   Ex:
   ```
   int largest(int array[], int n)
   {
   - - - - - -
   - - - - - -
   - - - - - -
   }
   ```

Example:-

```
/*Program to find largest element in an array */

#include<stdio.h>
Void main()
{
    int largest(int a[],int n); /*Function declaration */
    int value[5] = { 10, 15, 7, 25, 13};
    int large;
    Clrscn();
    large = largest(value, 5);
    printf("\n The largest element is: %d", large);
    getch();
}
int largest(int a[], int n)
{
    int i, max;
    max=a[0];
    for(i=1;i<n;i++)
    {
        if(max<a[i])
            max=a[i];
    }
    return max;
}
```

Explanation:-

When a function largest(value,5) is called, all elements of array 'value become the corresponding elements of array 'a' in the called function. The largest function finds the largest value in the array and returns the result to the main function.

TWO DIMENSIONAL ARRAYS TO FUNCTIONS:-

⇒Like one-dimensional arrays, it is also possible to pass 2D-arrays to Functions

RULES:-

1. In the function definition, we must indicate that the array has two-dimensions by including two sets of brackets. The size of second dimension must be specifi

Ex: int SumOfMatrix(int x[][N], int M, int N)
{
- - - - - -
- - - - - -
}

2. The function declaration should be similar to function header.

Ex: int SumofMatrix(int x[][N], int M, int N);

3. The function must be called by passing only the array name and the size of two dimensions.

Ex: int a[2][2] = {{1,2},{3,4}};
Sum = SumofMatrix(a, 2, 2);

Example program:-

```
/*Program to calculate Sum of all elements in a matrix*/
#include<stdio.h>
Void main()
{
    int SumofMatrix(int x[][N], int M, int N);
    int a[2][2] = {{1,3},{4,6}};
    int sum;
    clrscr();
    sum = SumofMatrix(a, 2, 2);
    printf("\n Sum of all elements in a matrix is : %d", sum);
    getch();
}
int SumofMatrix(int x[][N], int M, int N)
{
    int i, j, S=0;
    for(i=0; i<M; i++)
    {
        for(j=0; j<N; j++)
        {
            S = S + x[i][j];
        }
    }
    return S;
}
```

OUTPUT:-

Sum of all elements in a matrix is: 14

Example:-

```
/* Program to arrange array elements in ascending order */
#include<stdio.h>
void main()
{
  void sort(int x[], int m);
  int a[5] = {40, 90, 73, 81, 35};
  int i;
  clrscr();
  printf("\n Marks before sorting:");
  for(i=0; i<5; i++)
    printf("\t %d", a[i]);
  sort(a, 5); /* Function call (we are passing array name and its size */
  printf("\n Marks after sorting:");
  for(i=0; i<5; i++)
    printf("\t %d", a[i]);
  getch();
}
void sort(int x[], int m)
{
    int i, j, temp;
    for(i=0; i<m; i++)
    {
      for(j=i+1; j<m; j++)
      {
        if(x[i] > x[j])
        {
          temp = x[i];
          x[i] = x[j];
          x[j] = temp;
        }
      }
    }
}
```

OUTPUT:-

```
Marks before sorting:
  40  90  73  81  35
Marks after sorting:
  35  40  73  81  90
```

⇒ From the above program it is clear that, if a function changes the values of an array, then these changes will be made to the original array that passed to the function.

In C, array name represents the address of its first element. When an entire array is passed as an argument, the information about the addresses of array elements are passed to the called function. Therefore the called function refers to the original array stored in the memory. Therefore any changes made to the array elements will be reflected in the original array.

PASSING PARAMETERS TO FUNCTIONS

⇒ Parameters refer to the input given to a function.

⇒ The technique used to pass data from one function to another function is known as 'parameter passing'.

⇒ There are 2 ways to pass the parameters:

1. Pass by value (or) Call by value
2. Pass by address (or) Call by address (or) Call by reference (or) Pass by reference (or) Pass by pointers

1. Call by value (or) Pass by value:-

⇒ In call by value, the values of actual parameters are copied to the formal parameter list of the called function.

⇒ The called function works on the copy (i.e. on formal parameter list), but not on the original values of the actual parameters.

⇒ Any changes made to the formal parameters doesn't effect the actual parameters. This ensures that the original data in the calling function can't be changed accidently.

⇒ Changes made in the formal parameters are local to the block of called function.

Example:- Swapping of two values using call by value

```
#include<stdio.h>
Void main()
{
   void swap(int, int);
   int x, y;
   printf("Enter the values of x and y:");
   scanf("%d %d", &x, &y);
   swap(x,y); /* Calling the function by passing x,y values */
   printf("In main() function x=%d, y=%d", x, y);
   getch();
}
Void swap(int a, int b)
{
    int c;
    c=a;
    a=b;
    b=c;
    printf("In swap() function x=%d, y=%d", a, b);
}
```

Output:-

```
Enter the values of x and y: 5  10
In swap() function x=10, y=5
In main() function x=5, y=10
```

2. Pass by address (or) Call by address:-

⇒ In call by address, the memory addresses of the variables are passed to the called function. Function operates on addresses rather than values.

⇒ The called function directly works on the original data in the calling function

⇒ Any changes made in the formal parameters effect the actual parameters.

⇒ Here formal parameters are pointers to the actual parameters.

RULES FOR CALL BY ADDRESS:-

1. In call by address, the actual arguments in the function call must be the addresses of variables. Ex: swap(&x, &y) → address of variables
2. The formal arguments in the function header and function prototype must be prefixed by *. Ex: void swap(int *a, int *b);
3. To access the value of an actual argument in the called function, we must use the corresponding formal argument prefixed by *.

Example:

```
/* Swapping of two values using call by address */
#include<stdio.h>
Void main()
{
    void swap(int *, int *);
    int x,y;
    clrscr();
    printf("Enter values of x & y:");
    scanf("%d %d", &x, &y); /* addresses are passed */
    swap(&x, &y);
    printf("In main() function x=%d, y=%d", x, y);
    getch();
}
Void swap(int *a, int *b)
{
    int *c;
    *c = *a;
    *a = *b;
    *b = *c;
    printf("In swap function x=%d, y=%d", a, b);
}
```

Output:

```
Enter values of x &y: 10 5
In swap() x=5, y=10
In main() x=5, y=10
```

⇒ Any changes made in the arguments are permanent.

THE SCOPE, VISIBILITY AND LIFETIME OF VARIABLES:-

STORAGE CLASSES:-

⇒ In C, all the variables have a data type and a storage class.

⇒ The storage class of a variable gives the information about:

1. The location of the variable in which it is stored
2. The default initial value of the variable, if the initial value is not specifically assigned.
3. Scope of the variable i.e. in which block the variable is available (or) visible
4. Life time of the variable i.e. how long the variable would be in active mode.

⇒ There are 4 types of storage classes:-

1. Automatic storage class
2. Static storage class
3. External storage class
4. Register storage class

1. AUTOMATIC STORAGE CLASS:-

⇒ Automatic variables are declared inside a function.

⇒ They are created when the function is called and destroyed automatically when the function is exited, hence the name automatic.

⇒ Automatic variables are local to the function in which they are declared. Because of this, automatic variables are also called local variables (or) internal variables.

⇒ To define a variable as automatic, the keyword auto is used.

Ex:
```
main()
{
  auto int m;
  auto float avg;
  .....
  .....
}
```

⇒ A variable declared inside a function, without any storage class is, by default an automatic variable.

Ex:
```
main()
{
  int m; /* Automatic variable */
  .....
  .....
}
```

⇒ By defining a variable as automatic storage class,

→ It is stored in memory.

→ The default value of the variable will be ganbage value.

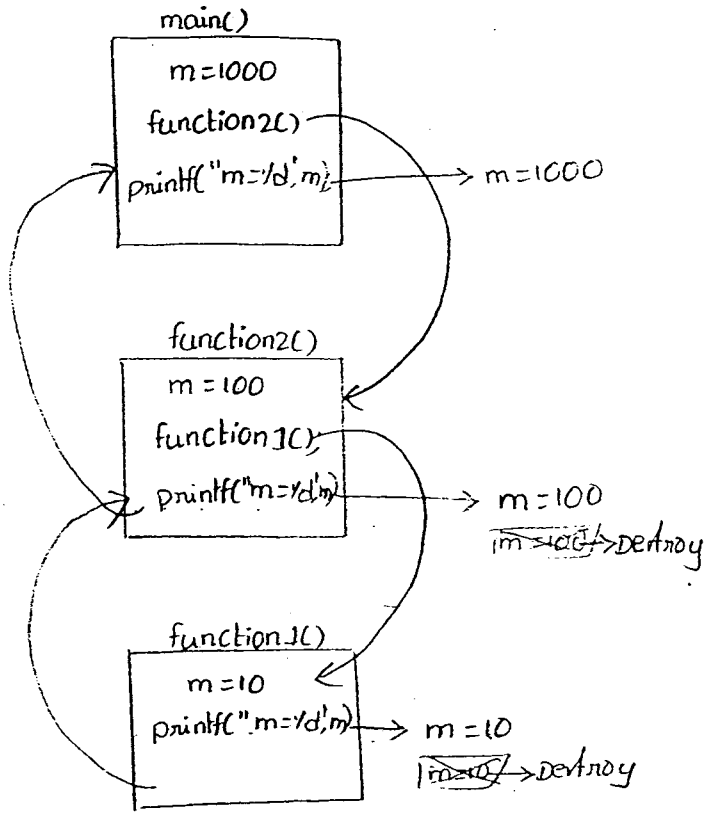
→ Scope of the variable is only within the function or block in which they are declared.

↳ The lifetime of the variable is until end of the function (or) block.

⇒ We may use the same variable name in different functions in the same progra without causing any confusion to the compiler.

Example program:-

```
void function1();
void function2();
main()
{
    int m=1000;
    function2();
    printf("m=%d",m);
}
function1()
{
    int m=10;
    printf("m=%d",m);
}
function2()
{
    int m=100;
    function1();
    printf("m=%d",m);
}
```



output: m=10
m=100
m=1000

Explanation:-

⇒ The main() function calls function2(), which in turn calls function1().

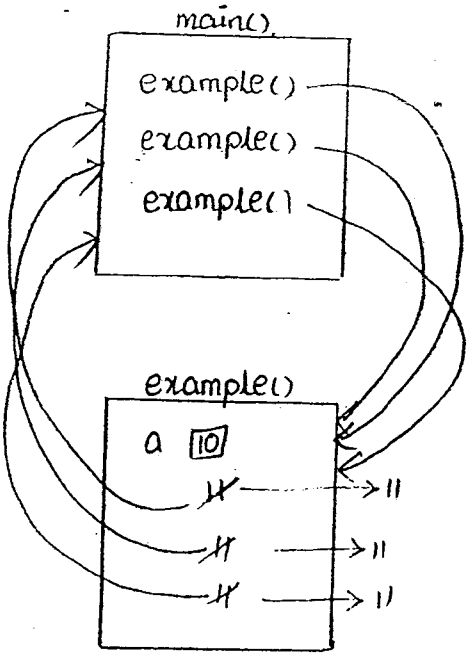
⇒ In function1() m=10, and this value is destroyed when it leaves the functio

⇒ In function2() m=100, and this value is destroyed when it leaves the functic

⇒ In main() function m=1000.

From this, it is clear that, the value of m is active when the control enters into the function, and destroyed automatically when the control leaves the function.

```
void example();
void main()
{
    example();
    example();
    example();
}
void example()
{
    auto int a;
    a=10;
    ++a;
    printf("\n %d", a);
}
```



OUTPUT:-
11
11
11

Explanation:-

⇨ The lifetime of the variable is only inside the function in which they are declared.

→ The main() function calls example()
→ When the function example() is called first time, the control goes to the example function a is initialized to '0'. Next 'a' is incremented by 1 (now a=11). This value is destroyed when the control leaves the function.
→ When example() is called second time, the control goes to the function definition initialize a=10 and increments its value by 1 (i.e. a=11) and this value is destroyed automatically when the control leaves the function

Example 3:-

```
void abc();
void main()
{
    abc();
    abc();
    printf("%d", a);
}
void abc()
{
    auto int a;
    a=30;
    ++a;
}
```

↳ Here 'a' is not accessible. Since the scope of the variable is only inside the function in which they are declared.

Output: Error → Undefined symbol 'a'.

REGISTER STORAGE CLASS:-

⇒ It is also possible to store variables in one of the machine's registers, instead of keeping them in the memory.

⇒ Register access is much faster than a memory access. So keeping the frequently accessed variables in the register will lead to faster execution of programs.

⇒ To define a variable as register storage class, the keyword register is used.

Example: register int x;
register char y;

⇒ When a variable is declared as a register variable,

→ It is stored in CPU Register.
→ The default value of the variable will be garbage value.
→ Scope of the variable is only within the function (or) block in which they are defined.
→ The lifetime of the variable is until end of function (or) block.

⇒ Only few variables can be placed in the registers. Most compilers allow only int or char variables to be placed in the register.

⇒ C will automatically convert register variables into non-register variable once the limit is reached.

Example program.

```
#include<stdio.h>
void main()
{
   register int i=1; /* i is declared as register variable */
   while(i<=10)
   {
     printf("\n%d",i);
     i++;
   }
   getch();
}
```

⇒ External variables are declared outside of all functions in the program.

⇒ External variables are also known as Global variables.

⇒ Global variables can be accessed by any function in the program.

⇒ When a variable is declared as global (or) extern

1. It is stored in memory.
2. The default value is initialized to zero
3. The scope of the variable is global.
4. The lifetime of the variable is until the program execution comes to an end.

Ex①:
```
int number;
float length;
main()
{
   ---------
   ---------
   number=10;
   length=1.5;
   ---------
}
function1()
{
   printf("%d", number);
}
function2()
{
   printf("%f", length);
}
```

Note: All the 3 functions can access the variables number & length.

⇒ If a local variable, and a global variable have the same name, the local variable will have highest precedence over the global one in which it is declared.

Ex②:-
```
int a=10;
void main()
{
   int a;
   a=1;
   printf("a=%d", a);
}
```

Output: a=1

⇒ When main() references the variable a, it will reference only its local variable, not the global one. So in main() function a=1.

⇒ Once a variable has been declared as global, any function can use it and change its value. Then the subsequent functions can reference only that new value.

Ex ③:-

```
void function1();
void function2();
void function3();
int x;
void main()
{
    x=10;
    printf("x=%d", x);
    function1();
    function2();
    function3();
}
void function1()
{
    x=x+10;
    printf("x=%d", x);
}
void function2()
{
    int x;
    x=1;
    printf("\n x=%d", x);
}
void function3()
{
        x=x+20;
        printf("\n x=%d", x);
}
```

OUTPUT:-

x=10
x=20
x=1
x=40

⇒ Another property of a global variable is that it is available only from the point of declarat to the end of the program.

Ex:

```
main()
{
    y=5;
    ----
    ----
    ----
}
int y; ⟶ /* Global variable */
function1()
{
    y=y+1;
}
```

Output: Error. The variable y is declared after th main() function. So it is not available in the main function

⇒ To avoid this, we have to specify y is a global variable explicitly by using the keywond extern

```
main()
{
    extern int y; ⟶ /* External declaration
    y=5;
    printf("y=%d", y);
    ----
}
int y; ⟶ /* Definition */
function1()
{
    y=y+1;
}
```

→ Although the variable y is defined after the ma function, the external declaration of y inside the function informs the compiler that y is a external variable.

Note: External declaration doesn't allocate storag space (i.e. memory) for variables.

⇒ A variable can be declared as static variable by using the keyword static.

Ex: static int x;
    static float y;

⇒ When a variable is declared as static,

1. It is stored in memory
2. The default value of the variable will be zero.
3. The scope of the variable is inside the function (or) block where it is defined.
4. The lifetime of the variable persists between function calls.

⇒ A static variable is initialized only once, when the program is compiled. It can't be reinitialized.

Example1:
```
#include<stdio.h>
void main()
{
   static int b;
   printf(" b=%d",b);
}
```
output: b=0. (∵ The default value of static variable is zero.)

Example 2:-
```
void start();
void main()
{
    start();
    start();
    start();
}
void start()
{
    static int x;
    x=x+1;
    printf("\n x=%d", x);
}
```
output:-
x=1
x=2
x=3

⇒ A static variable may be either an internal type (or) an external type depending on the place of declaration.

⇒ Internal static variables are those which are declared inside a function. The scope of internal static variables is up to the end of the function where it is defined.

⇒ An external static variable is declared outside of all functions and is available to all the functions in that program.

⇒ NOTE:-

Difference between a static external variable and a simple external variable is that static external variable is available only within the file where it is defined while the simple external variable can be accessed by other files.

NESTED BLOCKS:-

⇒ A set of statements enclosed in a set of braces is known as a block or Compound statement.

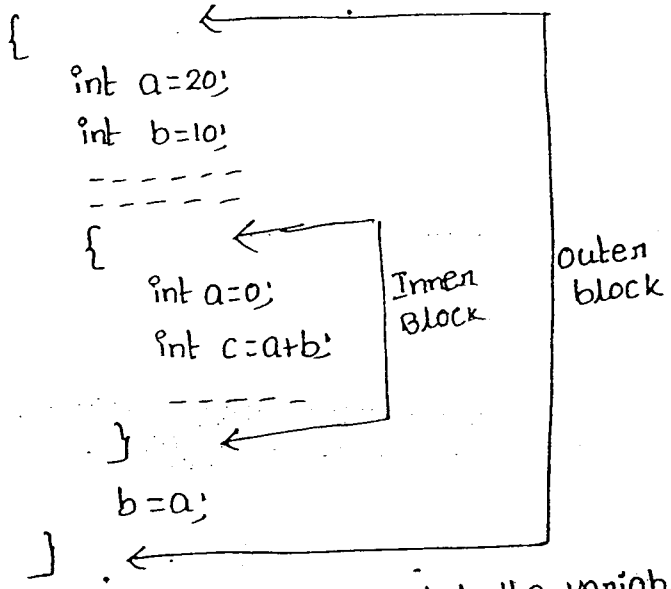
⇒ All the functions including the main() can use block of statements.

⇒ A block can have its own declarations and other statements.

⇒ It is also possible to have a block of statements inside the body of a function (or) another block.

Ex:

```
main()
{
    int a=20;
    int b=10;
    ------
    ------
    {
        int a=0;
        int c=a+b;
        -----
    }
    b=a;
}
```



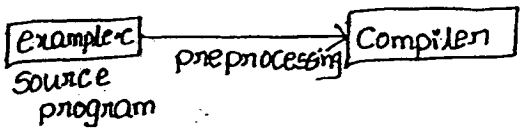
Note: When this program is executed, the variable c will be 10, not 30. The statement b=a; assigns b to 20, not zero.

PREPROCESSOR COMMANDS (OR) PREPROCESSOR DIRECTIVES:-

PREPROCESSOR COMMANDS:-

⇒ Preprocessor commands are the instructions that are executed before the source code passes through the compiler.

⇒ The program that process the preprocessor commands (or) directives is called a preprocessor.



⇒ Preprocessor directives are placed in the source program before the main() function.

⇒ Before the source code passes through the compiler, it is examined by the preprocessor for any preprocessor directives. If there are any preprocessor commands, appropriate actions are taken.

⇒ Each of the preprocessor directive begins with the symbol # and don't require a semicolon at the end.

⇒ Preprocessor commands can be placed anywhere in the program, but it is good practice to place them at the beggining.

Directive	Function
#define	– Defines a macro substitution
#undef	– Undefines a macro
#include	– Specifies the files to be included
#ifdef	– Test for a macro definition
#endif	– Specifies the end of #if
#ifndef	– Tests whether a macro is not defined.
#if	– Tests a compile-time condition
#else	– Specifies alternatives when #if test fails.

⇒ The preprocessor directives are broadly classified into 3 categories.

1. Macro substitution
2. File inclusion
3. Conditional compilation.

1. MACRO SUBSTITUTION DIRECTIVES:-

⇒ Macro substitution is a process where an identifier in a program is replaced by a predefined string

⇒ #define directive is used for this purpose.

Syntax: `#define identifier string`

If the above statement is included in the program, then the preprocessor replaces every occurance of the identifier in the source code by the string.

Identifier must be a valid name. String may be any text.

⇒ There are different forms of macro substitution:

1. Simple macro substitution
2. Argumented macro substitution
3. Nested macro substitution.

1. Simple macro substitution:

⇒ #define directive is used to define constant macros.

Ex: `#define PI 3.14`

During the preprocessing the preprocessor replaces every occurance of PI with 3.14

Ex:
```
#define  COUNT  100
#define  FALSE  0
#define  SIZE   50
```

Example program:-

```
#define A 10
void main()
{
    int i;
    i=A;
    printf("i=%d",i);
}
```

output: i=10

⇒ A macro definition can include more than one simple constant.

Ex:
```
#define AREA 5*25
#define SIZE 5+7
#define MAX sizeof(int)*4
```

Syntax: `#define identifier(f1,f2,....fn) String`

→ macro with arguments is known as macro call, which is similar to function call.

Ex: `#define CUBE(x) x*x*x`

→ Suppose if the following statement appears in the program

```
Volume = CUBE(3);
```

Then the preprocessor will expand this statement to volume = 3*3*3.

Example program:-

```
#include<stdio.h>
#define SQUARE(x) x*x
#define CUBE(x) x*x*x
void main()
{
    int x;
    x=SQUARE(2); /* x=2*2 */
    printf("Square of 2 is: %d", x);
    x=CUBE(2); /* x=2*2*2 */
    printf("Cube of 3 is: %d", x);
    getch();
}
```

3. Nested macro substitution:-

⇒ We can also use one macro in the definition definition of another macro.

Ex:
```
#define M 5
#define N M+1
```

Ex:
```
#define SQUARE(x) (x*x)
#define CUBE(x) (SQUARE(x)*x)
#define SIXTH(x) (CUBE(x)*CUBE(x))
```

⇒ For example the statement CUBE(x) #define SIXTH(x) is expanded to

```
(SQUARE(x)*x) * (SQUARE(x)*x)
```

Since SQUARE(x) is still a macro, it is further expanded into.

`((x*x)*x)*((x*x)*x)`, which finally evaluates x^6.

#undef:-

⇒A macro defined with #define directive can be undefined with #undef directive.

Syntax: #undef identifier

⇒It is useful when we don't want to allow the use of macros in any portion of the program.

Ex:
```
#include<stdio.h>
#define SIZE 100
void main()
{
   #undef SIZE
   int a[SIZE];
   int i;
   clrscr();
   for(i=1;i<SIZE;i++)
   {
     scanf("%d",&a[i]);
   }
}
```

FILE INCLUSION DIRECTIVE:-

⇒By using #include directive, we can include the files containing functions or macro definitions in the program so that no need to rewrite those function or macro definitions.

Syntax:
```
#include "filename"
   (OR)
#include <filename>
```

where filename is the name of the file containing the required functions.

→ The preprocessor inserts the entire contents of filename into the source code of the program.

⇒ When the filename is included within the double quotation marks, the search for the file is made in the current directory and then in the standard directories.

⇒ When the filename is included without double quotation marks, then the search for the file is made only in the standard directories.

```
#include<stdio.h>
#include <conio.h>
#include<math.h>
#include<stdlib.h>
#include<process.h>
#include<ctype.h>
#include<string.h>
```

CONDITIONAL INCLUSION DIRECTIVES:-

Some directives are used to compile a part of the program based on certain conditions.

The most commonly used conditional inclusion directives are:

1. #if
2. #else
3. #ifdef
4. #ifndef
5. #endif
6. #elif

1. #if :-

#if directive is used to check whether the result of a given expression is zero or not. If the result is not zero, then the statements after #if are compiled, otherwise the statements after #else are compiled.

Syntax:

```
#if expression
{
   statement1;
   statement2;
}
#else
{
   statement3;
   statement4;
}
#endif
```

Example:

```
#include<stdio.h>
void main()
{
  printf("Welcome to");
  #if 5>6
   printf("SITAMS");
  #else
   printf("CSE");
  #endif;
  getch();
}
```

#elif:-

```
#include<stdio.h>
#define AGE 10
void main()
{
   clrscr();
   #if AGE<=10
   printf("Child");
   #elif AGE>10 && AGE<=30
   printf("Youth");
   #elif AGE>30 && AGE<=60
   printf("Middle aged");
   #else
   printf("Old");
  #endif
}
```

#ifdef directive:-

#ifdef directive compile a part of the program only if the macro is defined as a parameter irrespective of its value.

Syntax:
```
#ifdef identifier
    statements1;
#else
    statement2;
#endif
```

Explanation: #ifdef directive checks whether the identifier is defined or not. If defined then the statements after #ifdef are compiled and executed otherwise the statements after #else are compiled and executed.

Example:

```
#include<stdio.h>
#define SIZE
void main()
{
   printf("Welcome to");
   #ifdef SIZE
   printf("Java");
   printf("C++");
   #else
   printf("C");
   #endif
}
```

output: Welcome to Java C++

```
#include<stdio.h>
void main()
{
   printf("Welcome to");
   #ifdef SIZE
   printf("C++");
   printf("Java");
   #else
   printf("C");
   #endif
   printf("Have a cool day");
}
```

output: Welcome to C. Have a cool day.

#ifndef directive:

⇒ #ifndef works exactly opposite to #ifdef directive.

⇒ This directive tests whether the identifier is defined or not. The statements after #ifndef are compiled and executed if identifier is not defined. If the the identifier is defined the statements after #else are compiled.

Example:

```
#include<stdio.h>
void main()
{
  clrscr();
  #ifndef T
  printf("macro is not defined");
  #else
   printf("macro is defined");
  #endif
  getch();
}
```

Output: macro is not defined.

MULTI FILE PROGRAMS:-

⇒ In real-time a program may use more than one source file which may be compiled seperately and linked later to form an executable file.

⇒ Multiple source files can share a variable.

⇒ Variables that are shared by two or more files are global variables. Therefore we must declare them as global variable in one file and then explicitly define them with extern in other files.

Example:-

file1.C

```
int m; /* global variable */
main()
{
  int i;
  m=10;
  ----
}
function1()
{
  int j;
  m=m+20;
  -----
  ---
}
```

file2.C

```
function2()
{
  extern int m;
  int j;
  -------
  -------
}
function3()
{
  int count;
  -----
  -----
}
```

⇒ The function2() in file2 can reference the variable m which is declared as global in file1. function3() cann't access variable m. However if the statement extern int m; is placed before all functions, then both the functions can refer to m.

⇒ The extern keyword tells the compiler that the variable types and names have already been declared somewhere else and no need to create storage space for them.

STRINGS

→ INTRODUCTION
→ DECLARAING AND INITIALIZING STRING VARIABLES
→ READING STRING FROM TERMINAL
→ WRITING STRINGS TO THE SCREEN
→ ARITHMETIC OPERATIONS ON CHARACTERS
→ PUTTING STRINGS TOGETHER
→ COMPARISION OF TWO STRINGS
→ STRING HANDLING FUNCTIONS
→ TABLE OF STRINGS
→ STRING/DATA CONVERSION

1. INTRODUCTION:-

String:- A string is a sequence of characters that is treated as a single item.

⇒ Any group of characters defined between double quotation marks is a string constant.

Ex: "Welcome"
"Well Done!"
"Hai"

⇒ The common operations performed on character strings include:

1. Reading and writing strings
2. Combining strings together
3. Copying one string to another
4. Comparing strings for equality
5. Extracting a portion of a string.

2. DECLARING AND INITIALIZING STRING VARIABLES:-

DECLARATION:-

⇒ String is a collection of characters. So strings can be declared as character arrays.

Syntax: `Char string_name[size];`

Where Char → is a data type.

String_name must be a valid identifier.

Size determines the number of characters in the string.

Ex: Char city[10];
Char name[30];
Char str[15];

⇒ The size should be equal to maximum number of characters plus 0' Because, the compiler automatically supplies a null character ('\0') at the end of the string.

INITIALIZATION:-

⇒ Character arrays may be initialized when they are declared.

⇒ In C, Strings may be initialized in either of the following 2 forms:

1. Char city[9] = "NEW YORK";

(OR)

2. Char city[9] = {'N','E','W',' ','Y','O','R','K'};

⇒ We can also initialize a character array without specifying the num of characters.

Ex: Char string[ ] = {'G','O','O','D','\0'};

In this case size of the array will be determined automatically b on the number of characters initialized. In the above example size the string is 5.

⇒ We can also declare the size much langer than the string size.

Ex: Char str[10] = "GOOD";

In this case, the computer creates a character array of size 10, places "GOOD" in it and initializes all other elements to NULL.

G	O	O	D	\0	\0	\0	\0	\0	\0

⇒ The size should be not greater less than the number of characters.

Ex: Char S[3] = "GOOD"; → is wrong, it will show a compile time er

⇒ We can't seperate the initialization from declaration:

Ex: Char S[5];
S = "Good"; is not allowed.

3. READING STRINGS FROM TERMINAL:-

⇒ There are 3 ways to read strings from terminal:

1. Using scanf() function
2. Using gets() function
3. Using getchar() function

1. Using scanf() function:

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

EXAMPLE:- Char city[10];
scanf("%s", city);

EXAMPLE:- Char S1[10], S2[10];
scanf("%s %s", S1, S2);

→ The scanf function automatically terminates the string that is read with null character.

NOTE: While reading strings using scanf() function, the & is not required before the variable name.

→ PROBLEM WITH SCANF():-

The problem with the scanf function is that it terminates its reading on the first white space it finds.

Ex: Char city[10];
scanf("%s", city);

For example, if the following line of text is entered

NEW YORK

then only the string "NEW" will be read into the string city, since the white sp after the string 'NEW' will terminate the reading of string

EXAMPLE PROGRAM:-

⇒ Write a program to read a series of words from terminal using scanf func

```
#include<stdio.h>
#include<conio.h>
void main()
{
  char word1[40], word2[20], word3[10];
  clrscr();
  printf("Enter 3 words:");
  scanf("%s", word1);
  scanf("%s %s", word2, word3);
  printf("\n word1 = %s \n word2 = %s \n word3 = %s", word1, word2, word3);
  getch();
}
```

OUTPUT: Enter 3 words: Oxford Road London

word1 = Oxford
word2 = Road
word3 = London

⇒ We can also specify the field width %ws in the scanf function for reading a spec number of characters from the input string.

EXAMPLE:- scanf("%ws", name);
↳ specifies number of characters to be read from the inpu str

Ex: char name[10];
scanf("%5s", name);

If the entered input string is "KRISHNAKUMAR" then the scanf function reads only 5 characters i.e. KRISH (∵ Because %5s → means it reads only 5 characte

READING A LINE OF TEXT USING SCANF FUNCTION:-

⇒ scanf with %s specification can't be used for reading more than one wor

Ex: char line[80];
scanf("%s", line); → Reads only a single word.

⇒ C supports a format specification known as edit set conversion code %[..], whic be used to read a line containing a variety of characters including whitesp

Ex: char line[80];
scanf("%[^\n]", line);
↳ Read a line of input from the keyboard.
printf("%s", line);

2. Using gets() function:-

⇒ gets() function can be used to read a line of text containing white spac

Syntax: `gets(string);`

gets() function reads characters into string from the keyboard until a new-line character (Enter button) is entered and adds a null character at end of the string.

Ex: char line[80];
gets(line);

EXAMPLE PROGRAM:-

⇒ Write a program to read a line of text using gets() function.

```
#include<stdio.h>
Void main()
{
    char name[30];
    clrscr();
    printf("Enter a string:");
    gets(name);
    printf("The string is: %s",name);
    getch();
}
```

INPUT:
Enter a string: C programming
The string is: C programming an

3. Using getchar() function:-

⇒ getchar() function reads a single character from the keyboard.

Ex: char ch;
ch=getchar();

⇒ We can use this function repeatedly to read a collection of characters from the terminal and store them into a character array. Thus a line of text can be read and stored in an array.

⇒ The reading is terminated when '\n' is entered and the null character is inserted at the end of the string.

```
Ex: char s[100],ch;
    int i=0;
    while((ch=getchar())!='\n')
    {
        s[i]=ch;
        i=i+1;
    }
    s[i]='\0';
```

EXAMPLE PROGRAM:-

⇒Write a program to read a line of text using getchar() function and display the

```
#include<stdio.h>
#include<conio.h>
Void main()
{
   Char Str[30],ch;
   int i=0;
   clrscr();
   printf("Enter a line of text:");
   while((ch=getchar())!='\n')
   {
      Str[i]=ch;
      i=i+1;
   }
   Str[i]='\0';
   printf("The entered text is: %s",Str);
   getch();
}
```

COPYING ONE STRING INTO ANOTHER STRING:-

⇒We cann't copy one string to another string directly.

Ex: Char String1[10], String2[10];

String1="ABC";
String2=String1 } are invalid.

⇒If we want to copy the string characters in string2 into String1, then we ca copy character-by-character basis.

EXAMPLE PROGRAM:

⇒Write a program to copy one String into another String.

```
#include<stdio.h>
Void main()
{
   Char String1[30],String2[30];
   int i;
   clrscr();
   printf("Enter a string:");
   scanf("%s",String1);
   for(i=0; String1[i]!='\0'; i++)
   {
      String2[i]=String1[i]; /*copying character by character */
   }
   String2[i]='\0';
   printf("After copying String2 is: %s",String2);
}
```

INPUT:-
Enter a string: Good

OUTPUT:-
After copying String2 is: Good.

WRITING STRINGS TO THE SCREEN:-

↳ Printing (or) Displaying strings

⇒ There are 3 ways to display strings on the screen.

1. Using printf() function
2. Using puts() function
3. Using putchar() function.

1. Using printf Function:-

⇒ The printf function with %s format can be used to print strings to the screen

Syntax: printf("%s", string_name);

The above function display the entire contents of string.

Ex: char city[10] = "NEW YORK";
printf("%s", city); → Displays "NEW YORK

EXAMPLE PROGRAM:-

⇒ Write a program to read a string using scanf function and display the same using printf function.

```
#include<stdio.h>
Void main()
{
    char str[30];
    clrscr();
    printf("Enter a string:");
    scanf("%s", str);
    printf("The entered string is: %s", str);
    getch();
}
```

INPUT:-
Enter a string: India
OUTPUT:-
The entered string is: India

⇒ We can also specify the precision with which array is displayed.

Syntax: printf("%w.ns", string);
↳ Number of characters to be displayed.
↳ Field width.

Ex: char name[15] = "United kingdom";

printf("%10.4s", name); → Display first 4 characters in the field width of 10 columns.
U n i t

printf("%-10.4s", name); → String will be printed left-justified
U n i t

printf("%15.0s", name); → Nothing will be printed
printf("%.3s", name); → Displays Uni
printf("%s", name); → Displays United kingdom

⇒ `printf("%*.*s", w, n, string);` → Display the first n characters of the string in the field width of w.

EXAMPLE PROGRAM:-

⇒ Write a program to print the following input

```
C
CP
CPR
CPRO
CPROG
CPROGR
CPROGRA
CPROGRAM
CPROGRA
CPROGR
CPROG
CPRO
CPR
CP
C
```

```
#include<stdio.h>
#include<conio.h>
void main()
{
    char string[10] = "CPROGRAM";
    int i, n=0;
    clrscr();
    for(i=0; string[i]!='\0'; i++)
    {
        n=i+1;
        printf("%.*s", n, string);
    }
    for(i=7; i>=0; i--)
    {
        n=i-1;
        printf("%.*s", n, string);
    }
    getch();
}
```

2. Using puts() function:-

⇒ puts function can be used to print strings to the screen.

Syntax: `puts(string_name);`

The puts function displays the contents of the string and then moves the cursor to the beginning of the next line on the screen.

Example:- `Char city[10]="PARIS";`

`puts(city);` → Display PARIS on the screen.

EXAMPLE PROGRAM:-

⇒ Write a program to read a line of text using gets() function and displa the same using puts() function.

```
#include<stdio.h>
#include<conio.h>
Void main()
{
  Char book[50];
  Clrscr();
  printf("Enter a book name:");
  gets(book); /*Reading using gets */
  puts(book); /*Displaying using puts*/
  getch();
}
```

INPUT:
Enter a book: C and Data Structure
OUTPUT: C and Data Structures

3. Using putchar() function:-

⇒ putchar function displays one character at a time on the screen.

Ex: `Char c='A';`
`putchar(c);` → Display A

⇒ We can use this function repeatedly to display a line of text stored in a character array using a loop.

Syntax:
```
Char city[10]="NEW YORK";
int i;
for(i=0;city[i]!='\0';i++)
{
  putchar(city[i]);
}
```

(OR)
```
Char city[10]="NEW YORK";
int i=0;
while(city[i]!='\0')
{ putchar(city[i]);
  i++;
}
```

↳ Displays one character at a time. i.e. it display the string character by characte

EXAMPE PROGRAM:

⇒ Write a program to display a line of text using getchar function and display th same using putchar function.

```
#include<stdio.h>
#include<conio.h>
void main()
{
   char str[50],ch;
   int i=0;
   clrscr();
   printf("Enter a string:");
   while((ch=getchar())!='\n')
   {
       str[i]=ch;
       i=i+1;
   }
   str[i]='\0';
  printf("The entered string is:");
   i=0;
   while(str[i]!='\0')
   {
      putchar(str[i]);
      i=i+1;
   }
   getch();
}
```

ARITHMETIC OPERATIONS ON CHARACTERS:-

⇒ Whenever a character variable is used in an expression, it is automc converted into an integer value (i.e. the equivalent ASCII value) by the system.

EXAMPE: int x;

x='a'; → convents 'a' into 97 i.e. ASCII value of a is 97

printf("%d", x); → Display 97 on the screen.

⇒ It is also possible to perform arithmetic operations on character constants and variables.

Ex: x = 'z'-1 = 122-1 = 121

y = 'u'+10 = 117+10 = 127

z = 'a'*2 = 97*2 = 194

a = 'y'+'z' = 121+122 = 243

ALPHABETS	ASCII VALUES
a-z	97-122
A-Z	65-90
0-9	48-57

⇒ Write a program to print the alphabet set a to z and A to Z in character and decimal form. i.e.

A-65 a-97
B-66 b-98
C-67 C-99
⋮ ⋮
Z-90 z=122

```
#include<stdio.h>
#include<conio.h>
void main()
{
    Char c;
    for(c=65; c<=90; c++)
    printf("\t %c - %d", c, c);
    printf("\n");
    for(c=97; c<=122; c++)
    printf("\t %c - %d", c, c);
    getch();
}
```

⇒ We may also use character constants in relational expressions.

Ex: Ch >= 'A' && Ch <= 'Z'

EXAMPLE PROGRAM:-

⇒ Write a program to check whether the given character is an uppercase letter or not.

```
#include<stdio.h>
void main()
{
    char ch;
    clrscr();
    printf("Enter a character:");
    scanf("%c", &ch);
    if( ch>='A' && ch<='Z')
     printf("The given character is an uppercase letter");
    else
     printf("The given character is not an uppercase letter");
    getch();
}
```

⇒ We can convert a character digit to its equivalent integer valu by using the following formula:

$$x = \text{character digit} - '0'$$

Ex: To convert '7' to its equivalent integer value

$x = '7' - '0'$

= ASCII value of '7' − ASCII value of '0'

= 55 − 48

= 7.

⇒ We can convert a string of digits into their integer value using function.

Syntax: atoi(string);

Ex:
```
char number[5] = "1988";
int year;
year = atoi(number);
year = 1988
```
→ converts "1988" (string) into 1988 →

6. PUTTING STRINGS TOGETHER:-

⇒ It is not possible to join (or) combine (or) add two strings directly by the simple arithmetic addition.

i.e. string3 = string1 + string2 ; → is not valid.

⇒ To combine string1 and string2, the characters from string1 and string2 should be copied into string3 one after the other.

⇒ The process of combining two strings together is called concatenation.

EXAMPLE PROGRAM:-

⇒ Consider the name of a person are stored in 3 arrays, namely first_name, second_name and last_name. Write a program to concatenate the 3 parts into one string called name.

```
#include<stdio.h>
void main()
{
  char first_name[10]="Viswanath";
  char second_name[10]="Pratap";
  char last_name[10]="Singh"; char name[30];
  int i,j,k;
  clrscr();
  for(i=0; first_name[i]!='\0'; i++)
  {
     name[i]=first_name[i]; → /* copying first_name into name */
  }
  name[i]=' ';
  for(j=0; second_name[j]!='\0'; j++)
  {
     name[i+j+1]=second_name[j]; → /*copying second_name into name*/
  }
  name[i+j+1]=' ';
  for(k=0; last_name[k]!='\0'; k++)
  {
     name[i+j+k+2]=last_name[k]; → /*copying last_name into name*/
  }
  name[i+j+k+2]='\0';
  printf("Full name is: %s", name); → /*Displaying full name i.e. name*/
}
```

7. COMPARISION OF TWO STRINGS:-

⇒ C doesn't support comparision of two strings directly.

ie. if (string1 == string2) is not permitted.

⇒ If you want to compare any two strings, then compare the two strings character by character.

⇒ The comparision is done until there is a mismatch on one of the strin terminates into a NULL character(\0).

EXAMPLE PROGRAM:-

⇒ Write a program to compare two strings. If both strings are equal, display th message "strings are equal". Otherwise display "strings are not equal.

```
#include<stdio.h>
#include<conio.h>
void main()
{
   char str1[10], str2[10];
   int i=0;
   clrscr();
   printf("Enter string1:");
   gets(str1);
   printf("Enter string2:");
   gets(str2);
   while(str1[i]==str2[i] && str1[i]!='\0' && str2[i]!='\0')
   {
      i=i+1;
   }
   if(str1[i]=='\0' && str2[i]=='\0')
   printf("Strings are equal");
   else
   printf("Strings are not equal");
   getch();
}
```

INPUT:

Enter string1 : Hello

Enter string2 : Hello

OUTPUT:

Strings are equal.

8. # STRING-HANDLING FUNCTIONS

⇨C-library supports a large number of string-handling functions that can be used to perform the following operations on strings:

- → Combining strings together
- → Copying one string to another
- → Comparing strings for equality
- → Finding length of a string
- → Reversing a string

1. Strcat() Function:-

⇨The strcat function joins two strings together.

Syntax: `Strcat(string1, string2);`

→ string1 and string2 are character arrays

→ Strcat function appends (i.e. adds) string2 to the end of string1.

→ The size of string1 must be large to hold the final string.

Ex:
```
Char Str1[10] = "good";
Char Str2[10] = "news";
Strcat(Str1, Str2);
```

The above function adds Str2 to the end of Str1.

∴ The content of Str1 is: goodnews

EXAMPLE PROGRAM:-

```
#include<stdio.h>
void main()
{
   Char Str1[10] = "very";
   Char Str2[10] = "good";
   clrscr();
   Strcat(Str1, Str2);
   printf("After concatenation string1 is: %s", Str1);
   getch();
}
```

OUTPUT:

After concatenation string1 is: Verygg

Ex:
```
Char Str1[20] = "good";
Strcat(Str1, "Morning");
```
→ Str1: goodmorning

⇒ Strcat function can be nested.

Ex: Strcat(Strcat(String1, String2), String3);

The above function concatenates all the 3 strings together and the resultant string is stored in string1.

EXAMPLE PROGRAM:-

```
#include<Stdio.h>
Void main()
{
  Char S1[30]="Very", S2[10]="Good", S3[10]="News";
  ClrScr();
  Strcat(Strcat(S1,S2),S3);
  printf("S1 is: %S", S1);
  getch();
}
```

OUTPUT:-
S1 is: VeryGoodNews

2. Strncat() Function:-

⇒ The strncat function concatenate the left-most n characters of String2 to the end of String1.

Syntax: `Strncat(String1, String2, n);`

Ex: Char S1[10]="Bala", S2[10]="Gurusamy";
Strncat(S1, S2, 4);
The above function adds the first 4 characters of S2 to the end of S
∴ S1 is: BalaGuru

EXAMPLE PROGRAM:-

```
#include<Stdio.h>
Void main()
{
  Char S1[30]="Programming", S2[10]="In C and Java";
  ClrScr();
  Strncat(S1,S2,2);
  printf("After Concatenation S1 is: %S", S1);
  getch();
}
```

OUTPUT:- After Concatenation S1 is: ProgrammingIn

3. Strcmp() Function:-

⇒ The strcmp() function compares two strings.

If strings are equal it returns 0.

If strings are not equal, it returns the numeric difference between the first non-matching characters in the strings.

Syntax: `Strcmp(String1,String2);`

Here String1 and String2 may be string variables or string constants.

Ex: Char S1[10]="Hello", S2[10]="Hello";

```
Strcmp(S1,S2);
Strcmp(S1,"Hello");
Strcmp("Their","There");
```

EXAMPLE PROGRAM:-

⇒ Write a program to check whether the given strings are equal or not.

```
#include<stdio.h>
void main()
{
  char Str1[10], Str2[10];
  int n;
  printf("Enter String1:");
  gets(Str1);
  printf("Enter String2:");
  gets(Str2);
  n=Strcmp(Str1,Str2);
  printf(".");          ↳ Comparing
  if(n==0)
   printf("Both strings are equal");
  else
   printf("Strings are not equal");
}
```

INPUT:-

Enter String1 : Good morning

Enter String2 : Good morning

OUTPUT:-

Both Strings are equal.

4. Strncmp() Function:-

⇒ The strncmp() function compares the left-most n characters of S1 to S1 and returns : 1. 0 if they are equal

2. Negative number if String1 is less than String2

3. Positive number if String1 is greater than String2

Syntax: `Strncmp(String1,String2,n);`

EX: strncmp(S1,S2,4); → Compares first 4 characters of S1 to S2.

PROGRAM:-
```
void main()
{
  char S1[15]="good morning", S2[15]="good Evening";
  int n;
  clrscr();
  n=strncmp(S1,S2,4);
  if(n==0)
    printf("First 4 characters are equal");
  else
    printf("First 4 characters are not equal");
  getch();
}
```

OUTPUT:
First 4 characters are equ

5. strcpy() Function:-

⇒ The strcpy function copies the contents of one string to another.

Syntax: `strcpy(string1,string2);`

→ The strcpy function copies contents of string2 to string1.

→ String2 may be a character array or a string constant

EX1: `char city1[10], city2[10]="Delhi";`

`strcpy(city1,city2);` → Copies contents of city2 to city1.
∴ city1 contains Delhi.

EX2: `strcpy(city1,"Hyderabad");` → Copies "Hyderabad" to city1.

PROGRAM:-
```
#include<stdio.h>
void main()
{
  char city1[10], city2[10]="DELHI";
  clrscr();
  strcpy(city1,city2);
  printf("After copying city1 is: %s",city1);
  getch();
}
```

OUTPUT:-
After copying city1 is: DELHI

8. Strrev() Function:-

⇒ The strrev function reverse a given string.

Syntax: `strrev(string);`

Ex: `char *str="good";`

`strrev(str);` → Reverse the string str. i.e. doog

PROGRAM:-

⇒ Write a program to reverse a given string.

```
#include<stdio.h>
void main()
{
    char *city="Delhi";
    clrscr();
    printf("Before reverse city is: %s", city);
    strrev(city);
    printf("After reversing city is: %s", city);
    getch();
}
```

OUTPUT:-

Before reverse city is: De
After reversing city is: ih

9. Strstr() Function:-

⇒ The strstr() function can be used to locate a sub-string in a main st

Syntax: ~~strstr(main,sub);~~

`strstr(main-string, sub-string);`

The above function searches the main string to check whether the sub is present or not. If the sub string is present in the main string, it re the first occurance of the sub-string. Otherwise it returns a NULL poin

EXAMPLE PROGRAM:-

⇒ Write a program to see whether the sub string is present in the main str If it present display the position of sub string.

```
#include<stdio.h>
void main()
{
    char m[20], s[10], *x;
    clrscr();
    printf("Enter the main string:");
    scanf("%s", m);
    printf("Enter the sub string:");
    scanf("%s", n);
    x=strstr(m,s);
    if(x==NULL)
    printf("Sub string is not present
                                the mai
    else
    printf("Position of substring in
            main string is: %d", x
    getch();
}
```

6. Strncpy() Function:-

⇨ The strncpy function copies only the left-most n characters of the source string to the target string variable.

Syntax: `strncpy(String1, String2, n);`

The above function copies the first n characters of string2 into str

EX: `strncpy(S1, S2, 5);` → Copies first 5 characters of S2 into S1.

`S1[6] = '\0';` → We have to explicitly include the NULL character.

PROGRAM:-

```
#include<stdio.h>
void main()
{
    char S1[10], S2[20] = "Programming";
    clrscr();
    strncpy(S1, S2, 7);        → copies first 7 characters of S2 to S1
                                  S1 contains Program
S1[7] = '\0';
    printf("String1 is: %s", S1);
    getch();
}
```

OUTPUT:
String1 is: Program

7. Strlen() Function:-

⇨ The strlen function returns the number of characters in a string.

Syntax: `n = strlen(string);`

Where n is an integer variable, which receives the length of the string

EX: `char str[5] = "good";`
`int n;`
`n = strlen(str);` → n = 4, Since str contains 4 characters.

PROGRAM:-

```
#include<stdio.h>
void main()
{
    char str[20] = "Programming";
    int n;
    clrscr();
    n = strlen(str);
    printf("The length of the string is: %d", n);
    getch();
}
```

OUTPUT:-
The length of the string is: 11

INPUT:-
OUTPUT:-

Enter the main string : Welcome
Enter the sub string : come

OUTPUT:

The position of sub string in main string is : 4

String Palindrome:-

⇒Write a program to check whether the given string is palindrome or not.

```
#include<stdio.h>
#include<conio.h>
void main()
{
   char str[30];
   int i,j,n,c=0;
   clrscr();
   printf("Enter a string:");
   scanf("%s",str);
   n=strlen(str);
   for(i=0,j=n-1;str[i]!='\0';i++,j--)
   {
       if(str[i]==str[j])
       continue;
       else
       {
          c=1;
          break;
       }
   }
   if(c==0)
   printf("%s is a palindrome",str);
   else
   printf("%s is not a palindrome",str);
   getch();
}
```

TABLE OF STRINGS:-

⇨ Suppose consider, we want to store a list of the names of students i a class, list of the names of employees in a company, list of places

⇒ A list of names can be treated as a table of strings.

⇒ A two-dimensional character array can be used to store a table of stri

Ex: student[30][15] may be used to store a list of 30 names whos length is not more than 15 characters.

⇒ Consider the following table of strings:

B	a	n	g	l	o	r	e
B	o	m	b	a	y		
m	a	d	r	a	s		
I	n	d	i	a			
C	h	e	n	n	a	i	

⇒ The above table can be stored as follows:

```
Char city[5][15] = { "Banglore", "Bombay", "madras", "India", "Chen
```

→ city[i] denotes the name of ith city.

For example city[0] denotes "Banglore", city[1] denotes "Bombay" so on.

EXAMPLE PROGRAM:-

⇒ Write a program to read and display a table of strings.-

```
#include<stdio.h>
void main()
{
  Char student[30][15];
  int i;
  clrscr();
  printf("Enter 30 students names:");
  for(i=0; i<30; i++)
  {
    scanf("%s", student[i]);
  }
  printf("Students names are:");
  for(i=0; i<30; i++)
  {
    printf("\n %s", student[i]);
  }
  getch();
}
```

Strstr():

```
#include<stdio.h>
#include<conio.h>
void main()
{
char main[10]="usha rani",sub[5]="rani",*x;
clrscr();
x=strstr(main,sub);
if(x==NULL)
printf("The sub string is not present in main
string");
else
printf("\nPosition of substring in main string
is:%d",(x-main));
getch();
}
```

Strrev():

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
char *s1="sitams";
clrscr();
printf("Before reverse s1 is:%s",s1);
strrev(s1);
printf("\nAfter reverse s1 is:%s",s1);
getch();
}
```

STRING PALINDROME (USING STRING H/ FUNC

```
void main()
{
  Char Str1[30], Str2[30];
  int n;
  Clrscr();
  Printf("Enter a string:");
  Scanf("%s", Str1);
  StrCpy(Str2, Str1);
  n = StrCmp (Str1, Strrev(Str2));
  if(n==0)
  Printf("%s is palindrome", Str1
  else
  Printf("%s is not a palindrome
  getch();
}
```

Without using string handling func.

```
main()
{
  Char Str[30];
  int i,j,n, c=0;
  Clrscr();
  Printf("Enter a string:");
  Scanf("%s", Str);
  n = Strlen(Str);
  for(i=0,j=n-1; Str[i]!='\0'; i++,
  {
      if(Str[i]==Str[j])
      continue;
      else
      {
        c=1;
        break;
      }
  }
  if(c==0)
  Printf("%s is palindrome", St
  else
  Printf("%s is not a palindrom
  getch()
}
```

⇒ WAP using pointer to print array of 5 strings.

```
#include<stdio.h>
#include<conio.h>
void main()
{
  char *name[5]={"India","Hydenabad","Chennai","Chittoor","Banglore"};
   int i;
   clrscr();
   for(i=0;i<5;i++)
   printf("\n%s",name[i]);
   getch();
}
```

⇒ Write an example program to return multiple values in a function?

```
main()
{
   void change(int *x,int *y,int *z);
    int a=10, b=20, c=30;
    clrscr();
    change(&a,&b,&c);
    printf("a=%d\n b=%d\n c=%d", a, b, c);
    getch();
}
void change(int *x,int *y, int *z)
{
    x=x+1;
    y=y+1;
    z=z+1;
}
```

String comparison without using string handling functions:

```
#include<stdio.h>
#include<conio.h>
void main()
{
char str[10],str1[10];
int i=0;
clrscr();
printf("Enter string1: ");
gets(str);
printf("Enter string2:");
gets(str1);
while(str[i]==str1[i]) /*&& str[i]!='\0' &&
str1[i]!='\0')*/
i++;
if(str[i]=='\0' && str1[i]=='\0')
printf("Strings are equal");
else
printf("Strings are not equal");
getch();
}
```

String comparison using string handling functions:

```
#include<stdio.h>
#include<conio.h>
void main()
{
char str1[10],str2[10];
int x;
clrscr();
printf("Enter sring1:");
gets(str1);
printf("Enter string2:");
gets(str2);
x=strcmp(str1,str2);
if(x==0)
printf("Both strings are equal");
else
printf("Strings are not equal");
getch();
}
```

String Concatenation using string handling functions:

```
#include<stdio.h>
#include<conio.h>
void main()
{
char str1[10]="very";
char str2[10]="good";
clrscr();
strcat(str1,str2);
printf("\nAfter concatenation String1
is:%s",str1);
getch();
}
```

Example2:

```
#include<stdio.h>
#include<conio.h>
void main()
{
char str1[10]="very";
clrscr();
strcat(str1,"bad");
printf("\nAfter concatenation String1
is:%s",str1);
getch();
}
```

Example 3:

```
#include<stdio.h>
#include<conio.h>
void main()
{
char
str1[20]="Very",str2[5]="good",str3[5]="news";
clrscr();
strcat(strcat(str1,str2),str3);
printf("After concatenation string1 is:%s",str1);
getch();
}
```

Example 4:

```
#include<stdio.h>
#include<conio.h>
void main()
{
char str1[10],str2[5];
clrscr();
printf("Enter string1:");
gets(str1);
printf("Enter string2:");
gets(str2);
strcat(str1,str2);
puts(str1);
puts(str2);
getch();
}
```

String Copy using string handling functions:

```
#include<stdio.h>
#include<conio.h>
void main()
{
char str1[10]="good",str2[10]="Morning";
clrscr();
strcpy(str1,str2);
printf("\n After String copy string1 is:%s",str1);
getch();
}
```

String Copy without using string handling functions:

```
#include<stdio.h>
#include<conio.h>
void main()
{
char str1[20]="good morning",str2[20];
int i;
clrscr();
for(i=0;str1[i]!='\0';i++)
str2[i]=str1[i];
str2[i]='\0';
printf("After copying string2 is:%s",str2);
getch();
}
```

String length using string handling functions:

```
#include<stdio.h>
#include<conio.h>
void main()
{
char str[10]="very good";
int l;
clrscr();
l=strlen(str);
printf("\n The given string is:%s",str);
printf("\nLength of the string is:%d",l);
getch();
}
```

String length without using string handling functions:

```
#include<stdio.h>
#include<conio.h>
void main()
{
char str[10]="Very good";
int i=0;
clrscr();
while(str[i]!='\0')
i=i+1;
printf("\n The length of the string is:%d",i);
getch();
}
```

Strncpy():

```
#include<stdio.h>
#include<conio.h>
void main()
{
char s1[10]="very good",s2[10];
clrscr();
strncpy(s2,s1,4);
s2[4]='\0';
printf("String2 is:%s",s2);
getch();
}
```

Strncmp()

```
#include<stdio.h>
#include<conio.h>
void main()
{
char s1[15]="sitams cse",s2[15]="sitams ece";
int x;
clrscr();
x=strncmp(s1,s2,5);
if(x==0)
printf("First 5 characters are equal");
else
printf("Not equal");
getch();
}
```

Strncat():

```
#include<stdio.h>
#include<conio.h>
void main()
{
char s1[20]="Bala",s2[10]="Gurusamy";
clrscr();
strncat(s1,s2,4);
printf("After concatenation s1 is:%s",s1);
getch();
}
```

Pointers:

→ A pointer is a memory variable that stores a memory address

→ Pointer can have any name as of other variable and it is declared in the same fashion like other vari but it is always denoted by '*' operator.

Features of Pointers:

1. Pointers save the memory space.
2. Execution time with pointer is faster becau the data is manipulated with the address ie direct access to memory location.
3. The memory is accessed efficiently with the pointe The pointers assigns memory space and it also releases Dynamically memory is allocated.
4. Pointers are used with data structures. They are useful to represent two dimensional and multidimensional ar

Pointer Declaration:

→ Pointer variables can be declared as follows.

Ex: `int *x;`
`float *f;`
`char *ch;`

→ In the first statement x is an integer pointer and it tells the compiler that it holds the address of any integ variable. In the same way f is a floating pointer and ch is character pointer that stores the address of fl and character variables respectively.

→ The indirection operator (*) is also called as deferencing operator. When a pointer is dereferenced, the value at that address stored by the pointer is retrieved.

→ Normal variable provides direct access to their values. Whereas pointer provides indirect access to value of variable whose address it stores

→ The indirection operator (*) is used in two distinct ways with the pointers declaration and deference.

→ When a pointer is declared, the star indicates that it is a pointer, not a normal variable.

→ when a pointer is ~~defe~~ dereferenced. the ~~star~~ ~~dec~~ indicate that value at that memory location stored in the pointer is to be accessed rather than address.

→ The indirection operator (*) is the same operator as multiplication operator. The Compiler knows which operator to call based on the Content.

→ The & is the address operator and it represents the address of the variable. The %u is used with printf() function for printing the address of a variable along with the & operator immediately proceeding with variable to return the address of variable which is a whole number

Example Write a program to display the value of variable and it location using pointers

```
#include <stdio.h>
#include <conio.h>
main()
{  int v=10, *p;
   clrscr();
```

```
printf("\n Address of v = %u", &v);
printf("\n value of v = %d", *p);
printf("\n Address of p = %u", &p);
getch();
}
```

output : Address of v = 4060
Value of v = 10
Address of p = 4062

Explanation:

→ In the above program v is an integer variable and its value is 10. The variable p is declared as pointer variable

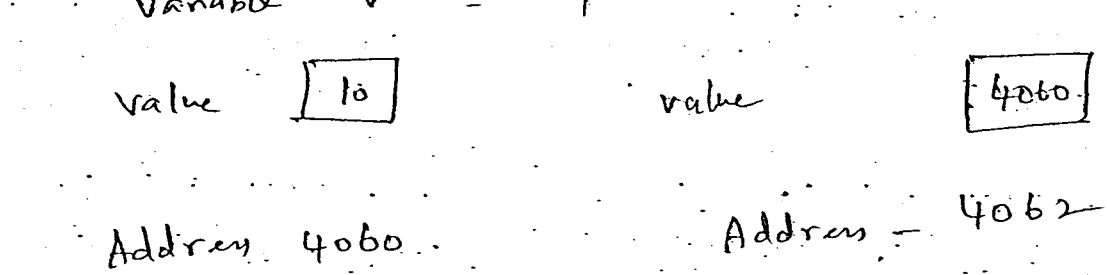
→ The statement p = &v assigns the address of v to p i.e p is the pointer to the variable v.

→ To access the address and value of v pointer 'p' can be used.

→ The value of p is nothing but address of variable v and display the value *p is used.

→ The pointer variable also have an address and are displayed using & operator.

→ The above explanation can be given by diagram as shown below.



Write a program to add two numbers through variable and their pointers

```
#include<stdio.h>
#include<conio.h>
main()
{ int a, b, c, d, *p1, *p2;
  clrscr();
 printf("Enter two numbers:");
 scanf("%d %d", &a, &b);
    p1 = &a;
    p2 = &b;
    c = a+b;
    d = *p1 + *p2;
 printf("\n Sum of A & B using variable: %d", c);
 printf("\n Sum of A & B using pointers: %d", d);
 getch();
}
```

output: Enter two numbers: 5 6
sum of A & B using variable: 11
sum of A & B using pointers: 11

Explanation:
→ In the above program sum of two numbers is obtained by two ways.
1. Adding variables a and b directly
2. Through pointer of a and b i.e p_1 and p_2 respectively
→ Address of a and b are stored in p_1 and p_2 respectively. Thus adding *p1 and *p2 gives the sum of a, b

Arithmetic Operations with pointers:

→ Arithmetic operation on pointer variables like increase, decrease, prefix and postfix operations can be performed

→ The increase of pointer variable for integer, the address is increased by two because integer requires 2 byte

→ Similarly for characters, floating point and double requires 1, 4 and 8 bytes respectively.

→ The following table shows the increase and decrease of pointer variable for all data type variables

Initial Data type	Address	operation		Address after operation		Require bytes
int i = 2	4046	++	--	4048	4044	2
char c = 'x'	4053	++	--	4054	4052	1
float f = 2.2	4058	++	--	4062	4054	4
double d = 2.6	4060	++	--	4068	4052	8

Write a program to show the effect of increment on pointer variables. Display the memory location of integer character and floating point numbers before and after increment of pointers

```
#include <stdio.h>
#include <conio.h>
void main()
{ int x, *x1;
  char y, *y1;
  float z, *z1;
```

```
clrscr();
printf("Enter integer, character, float values: ");
scanf("%d %c %f", &x, &y, &z);
x1 = &x;
y1 = &y;
z1 = &z;
printf("Address of x = %u\n", x1);
printf("Address of y = %u\n", y1);
printf("Address of z = %u\n", z1);
x1++;
y1++;
z1++;
printf("After Increment in pointers:\n");
printf("Address of x = %u\n", x1);
printf("Address of y = %u\n", y1);
printf("Address of z = %u\n", z1);
getch();
}
```

output: Enter integer, character, floating value: 2

Address of x = 4046
Address of y = 4053
Address of z = 4058
After Increment in pointers
Address of x = 4048
Address of y = 4054
Address of z = 4062

Explanation: In the above Program 4046 is the addr of integer x, 4053 for character value y and 4058 for float

→ On increasing of pointer the address of integer, character, and float will be 4048, 4054 and 4062 respectively.

→ This is because every time a pointer points immediate to or a previous location of its type after increase or decrease in the operation respectively.

Write a program to perform different arithmetic operation using pointers

```
#include<stdio.h>
#include<conio.h>
main()
{ int a=25, b=10, *p, *q;
  p=&a;
  q=&b;
  clrscr();
  printf("\n Addition of a+b = %d", *p+b);
  printf("\n Subtraction of a-b = %d", *p-b);
  printf("\n Product of a*b = %d", *p**q);
  printf("\n Division of a/b = %d", *p/(*q));
  printf("\n a Mod b = %d", *p%(*j));
  getch();
}
```

output: Addition of a+b = 35
Subtraction of a-b = 15
Product of a*b = 250
Division of a/b = 2
a Mod b = 5

Explanation

→ The various arithmetic operation can be perform on a pointer such as addition, subtraction, multiplicat division and mod.

→ The value stored at the address is taken for operation but not the address.

→ The arithmetic operations are impossible with addresses.

Ex: Two addres location are not possible to add.

Pointers and Arrays:

→ Array name itself is an address or pointer.

→ It points to the address of the first element of the array.

→ The elements of array together with their addresses can be displayed by using array name itself.

→ Array elements are always stored in contiguous memory location.

Write a program to display array element with their addre using array name as pointers

```
#include<stdio.h>
#include<conio.h>
main()
{ int x[5] = {2, 4, 6, 8, 10}, k=0;
  clrscr();
  printf("Element Number \t Element\t Address \n");
```

```
    while (k<5)
    {
        printf (" x[%d] = \t %d \t %u \n", k, *(x+k), x+k
        k++;
    }
}
```

output:-

Element Number	Element	Address
x[0]	2	4056
x[1]	4	4058
x[2]	6	4060
x[3]	8	4062
x[4]	10	4064

Explanation:

→ In the above program variable k acts as an element number and its value varies from 0 to 4.

→ When it is added with array name x ie with address of first element, it points to consecutive memory location.

→ Thus the element no., element and their address are displayed.

Write a program to find sum of all the elements of an array by using array name as a pointer.

```
#include<stdio.h>
#include<conio.h>
main()
{ int sum=0, i=0, a[]={1,2,3,4,5};
  clrscr();
```

```
printf ("Elements \t Values \t Address \n");
while (i<5)
{
    printf ("a[%d] \t %d \t %u \n", i, *(a+i), (a+i));
    sum = sum + *(a+i);
    i++;
}
printf ("Sum of Array elements = %d", sum);
getch();
}
```

output:

Elements	Values	Address
a[0]	1	4056
a[1]	2	4058
a[2]	3	4060
a[3]	4	4062
a[4]	5	4064

Sum of array elements = is

Explanation:

→ In this program array name 'a' act as pointer and variable 'i' is used for referring elem numbers

→ Using the while loop and expression *(a+i) and (a+i) various elements and their addresses are displayed respectively.

→ In the sum variable sum of all the elements is obtained.

Pointers and two dimensional arrays:

→ A matrix can represent two dimensional array. of an element where first argument is row numb and second as column number.

→ To display the elements of two dimensional array using pointers it is essential to have & operator as prefix with an array name followed by element number, otherwise compiler shows an error.

Write a program to display array elements and their address using pointers

```
#include <stdio.h>
#include <conio.h>
main()
{
    int i, j=1, *p;
    int a[3][3] ={{1,2,3}, {4,5,6}, {7,8,9}};
    clrscr();
    printf("Elements of an array with their address \n");
    p= &a[0][0];
    for(i=0; i<9; i++, j++)
    {
        printf("%5d [%5u] ", *p, p);
        p++;
        if(j==3)
        { printf("\n");
          j=0;
        }
    }
    getch();
}
```

output:

Elements of an array with their addresses.

1 [4052] 2 [4054] 3 [4056]

4 [4058] 5 [4060] 6 [4062]

7 [4064] 8 [4064] 9 [4064]

Explanation:

→ In the above program two dimensional arr is declared and initialized.

→ The base address of an array is assigned t integer pointer p, while assigning & operator is to b prefixed with array name followed by element number

→ The ~~statement~~ pointer p is printed and increased in for loop till it prints the entire array elements

→ The if statement splits a line when the elements in each row are printed.

Array of pointers:

→ Arrays of different standard data types such as array of int, float, character etc are possible in c.

→ In the same way 'c' language also supports arr of pointers which ~~before~~ is nothing but collection of addresses.

→ Here, we store address of variables for which we have to declare an array as a pointer.

Write a program to store addresses of different elements of an array using array of pointers.

```
#include <stdio.h>
#include <conio.h>
main()
{ int *arrp[3];
  int a[3] = {5, 10, 15}, k;
  clrscr();
  for(k=0; k<3; k++)
    { arrp[k] = a+k;
    }
  printf("Address \t Element \n");
  for(k=0; k<3; k++)
    { printf("\t%u", arrp[k]);
      printf("\t%d", *(arrp[k]));
    }
  getch();
}
```

arrp[0] = &a[0]
arrp[1] = &a[1]
arrp[2] = &a[2]

output:

Address	Element
4060	5
4062	10
4064	15

Explanation:
→ In the above program *arrp[3] is declared as an array of pointers.
→ using first for loop the address of various ele of array 'a' are assigned to *arrp[].
→ The second for loop pick up the addresses fr arrp[]. and display value present at that location along wi the address.

Pointers to Pointers:

→ Pointer is known as a variable containing address of another variable

→ The pointer variables also have an address.

→ The pointer variable containing address of another pointer variables is called as Pointer to Pointer.

→ This chain can be continued to any extent.

Write a program to print value of a variable through pointer and pointer to pointer.

variable	Pointer *	Pointer to pointer **
(4056) a	(4060) p	(4064) q
2	(4056)	(4060)

q = 4060
*q = 4056
**q = 2

```
#include<stdio.h>
#include<conio.h>
main()
{ int a=2, *p, **q;
  p = &a;
  q = &p;
  clrscr();
  printf("value of a = %d; Address of a = %u\n", a,
printf("Through *p the value of a = %d, Address of a = %u\n", *p, p
printf("Through **q the value of a = %d, Address of a = %u\n", **q, q
  getch();
}
```

output:

value of a = 2, Address of a = 4056.
Through *p value of a = 2, Address of a = 4056
Through **q value of a = 2, Address of a = 4056

Exploration:

→ In the above program variable p is declared as a pointer. The variable q is declared as pointer to another pointer.

→ The address of variable a is assigned to pointer p.

→ The address of pointer p is assigned to q.

→ The variable q contains the address of the pointer variable *p. Hence q is pointer to pointer.

→ The program displays value and address of variable a, my variable itself and pointers p and q.

Void pointers:

→ Pointers can also be declared as void type

→ Void pointers cannot be ~~def~~ dereferenced without explicit type conversion because being void the compiler cannot determine the size of the object that pointer points to.

→ Void pointer declaration is possible but void variable declaration is not allowed.

→ The declaration void p displays the error message "size of p is unknown or zero" after compilation

Write a program to declare as void pointer. Assign address of int, float and char variables to the void pointers using type casting method. Display the contents of various variables

```
#include<stdio.h>
#include<conio.h>
 int p;
 float d;
 char c;
void *pt = &p;
main()
{
 *(int*) pt = 12;
   printf("p = %d\n", p);
    pt = &d;
 *(float*) pt = 2.3;
   printf("d = %f\n", d);
   pt = &c;
 *(char*) pt = 'S';
   printf("c = %c", c);
 getch();
}
```

output: p = 12
d = 2.3
c = S

Explanation:

→ In the above example variable p, d and c are declared as int, float and char respectively

→ pointer pt is a pointer of type void

→ The pointer is initialized with address of integer variable p ~~the pointer 'p' points to variable~~

→ The statement *(int*) pt = 12 assigns the integer value 12 to pointed pt i.e. to variable p.

→ The declaration *(int*) tells the compiler the value assigned is of integer type.

→ ~~assignment of float and~~ char type are ca

Pointer to function:

→ In c language every variable has address except register variables. We can access the address of variable using pointers.

→ C function also have an address. ~~which we~~

→ ~~invoke~~ ~~using~~ we invoke the function using its address.

Write a program to display address of user defined function

```
#include <stdio.h>
# include <conio.h>
~~int~~ show (void);
main()
{
    clrscr;
     show();
    printf("%u", show);
}
show()
{ printf (" Address of function show() is :");
}
```

output: Address of function show() is : 530

Explanation: → In the above program the show() is a user defined function.

→ The function name without parenthesis displays the address of the function.

→ In the output the address of function show is displayed.

→ To assign the obtained address of function, we need to declare a pointer which can hold the address of the function.

→ The following statement declare the pointer that can hold the address of the function.

```
int (*p) ();
```

Here 'p' is the pointer name and the bracket indicates that it is a pointer to function.

Example: Write a program to call a function using pointer

```
#include <stdio.h>
#include <conio.h>
  show();
main()
{ int (*p) ();
   p = show;
   (*p) ();
  printf ("%u", show);
  getch();
}
 show()
 { clrscr();
  printf("Address of function show() is:");
 }
```

Output: Address of function show() is : 531

Explanation → In the above program variable p is pointer to function → Address of function show() is assigned to pointer p → Using function pointer the function show() is invoked and the output of program is displayed.

```
#include <stdio.h>
#include <conio.h>
main()
{
    int x[50], i, n;
    clrscr();
    printf("Enter the size of array:\n");
    scanf("%d", &n);
    printf("Enter the elements of array:");
    for(i=0; i<n; i++)
    {
        scanf("%d", (x+i));
    }
    printf("The elements of array are:");
    for(i=0; i<n; i++)
    {
        printf("%5d", *(x+i));
    }
    getch();
}
```

100 10
3
106
104

(20)

Write a program using pointer to read an array of integer and prints its elements in reverse order.

```
#include <stdio.h>
#include <conio.h>
main()
{
    int x[50], i, n, *y;
    clrscr();
    printf(" Enter the size of array : ");
    scanf("%d", &n);
    y = x;
    printf(" Enter the elements of array : ");
    for(i=0; i<n; i++)
      { scanf("%d", (y+i));
      }
    printf(" The elements of array in reverse order is
    for(i=n-1; i>=0; i--)
      {
        printf("%5d", *(y+i));
      }
    getch();
}
```

Write a c program ~~that~~ ~~way~~ ~~of~~ ~~pointer~~ to show that (11)
pointers of any datatype occupies same space.

```
#include<stdio.h>
#include<conio.h>
main()
{
    int *i;
    float *f;
    char *c;
    double *d;
    clrscr();
    printf("The size of integer pointer is %d\n", sizeof(i));
    printf("The size of character pointer is %d\n", sizeof(c));
    printf("The size of float pointer is %d\n", sizeof(f));
    printf("The size of double pointer is %d\n", sizeof(d));
    getche();
}
```

free function is used to free the memory allocation allocated by malloc, calloc and realloc when they are no longer in use.

→ The declaration of free function is

```
void free (void *ptr);
```

Dynamic Memory allocation:

→ In static memory allocation, memory is allocated to the variables and arrays at compile time

→ Allocating memory to the object at runtime is called as dynamic memory allocation.

Memory Management functions:

→ four types of memory have been defined for dynamic memory allocation.

1. malloc()
2. calloc()
3. realloc()
4. free()

→ First 3 functions are used to allocate memory and last one is to free the memory that is not in use

→ All these function are included in the standard library function, stdlib.h.

Malloc():

→ malloc function is used to allocate a block of memory and it returns void pointer to the address of first byte in the memory.

→ The declaration of malloc function is as follows:

```
void *malloc(size_t size);
```

→ size_t represents the type, it is usually an unsigned integer.

→ The size of the malloc's actual parameters is defined by using sizeof operator.

Example:

int m = void *malloc (sizeof (int))

→ The malloc function with zero size may return a null pointer or some other value.

Calloc:

→ Calloc function is used to allocate memory dynamical for arrays.

→ The only difference between calloc and malloc is that calloc sets memory to null character but malloc doesn't

→ Declaration of calloc is as follows.

void *calloc (size_t element_count, size_t elem size);

calloc function with zero size may return a null pointer.

Ex: int .c[] = (int *) calloc (50, sizeof (int));

→ Here array of 50 integer is allocated.

Realloc:

→ Realloc resizes the block of memory either by deleting or extending the memory block

→ It allocates new block, if the existing block cannot be extended and copies the existing memory allocation to new block and deletes the old one.

void *realloc (void *ptr, size_t newsize);

free:

→ free function is used to free the memory allocation allocated by malloc, calloc and realloc when they are no longer in use.

→ The declaration of free function is

```
void free (void *ptr);
```

UNIT-4

STRUCTURES, UNIONS and FILES

Structures and Unions:

→ Introduction
→ Defining a structure
→ Declaring structure variables
→ Accessing structure members
→ Structure initialization
→ Copying and comparing structure variables
→ Operations on individual members
→ Arrays of structures
→ Arrays within structures
→ Structures within structures
→ Structures and functions
→ Unions
→ Size of structures
→ Bit fields
→ Type def
→ Enum

1. INTRODUCTION:- (OR) SIGNIFICANCE OF STRUCTURES (OR) NEED FOR STRUCTURES:-

⇒ A variable can store a single value at a time.
⇒ Arrays can be used to represent a collection of similar data items.
⇒ We can't use arrays if we want to represent a collection of different data items using a single name.
⇒ For example consider, we want to maintain employee information.
→ Employee information may include employee name, age, qualification, salary etc.
→ To maintain employee information different data types are required.
→ Name and qualification are char data type, age is integer and salary is float.
→ All these data types can't be represented in a single array. We need to declare different arrays for each data type. Hence, arrays can't be useful here.
→ For handling these mixed data types, C provides a feature known as structures

DEFINITION:-

⇒A structure is a collection of hetrogeneous elements.

(OR)

⇒A structure is a collection of different data items grouped under a single name.

EXAMPLES (OR) USES:-

⇒Structures can be used to represent the following:

1. Customer details : name, telephone, city, category
2. Address : name, door-number, street, city
3. Employee details : name, age, qualification, salary.
4. City : name, country, population
5. Book details : author, title, price, year
6. Date : day, month, year
7. Time : seconds, minutes, hours

⇒By using structures, the data can be organized in a more meaningful way.

FEATURES OF STRUCTURES:-

1. Nesting of structures is possible. ie. one can create structure within a structure. Using this feature one can handle complex data types.
2. To copy elements of one array to another array, the elements are copied one by one. It is not possible to copy all the elements at a time.

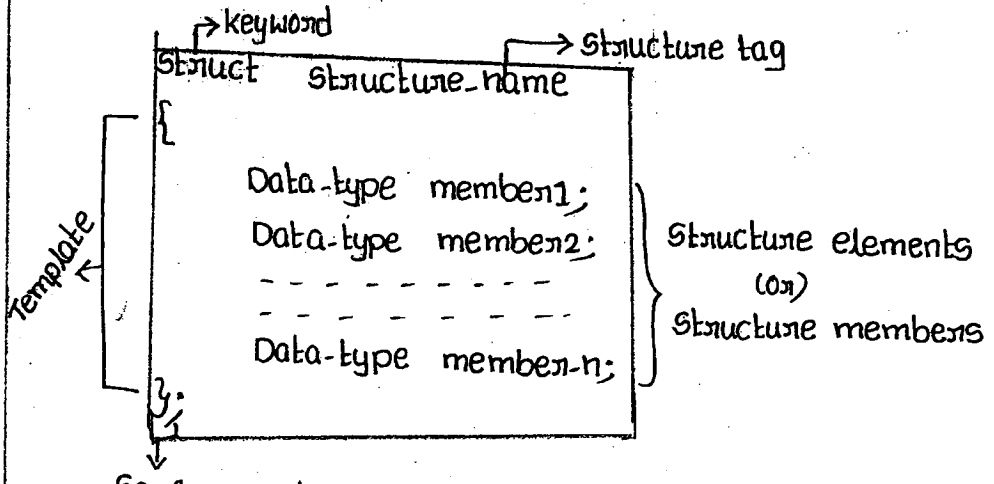
   Where as in structure, it is possible to copy the all the structure elements at a time using = operator.
3. It is also possible to pass structure elements to a function.
4. It is also possible to create structure pointers.

2. DEFINING A STRUCTURES:-

⇒ Each and every structure must be defined and declared before they are used in a program.

⇒ Structure definition creates a format that may be used to declare structure variables.

Syntax: The general format of structure definition is as follows:



→ Structure declaration always starts with struct keyword.

→ Structure_name is known as a tag, it may be used to declare structure variables. Structure name is optional in some situations.

→ The structure definition must be enclosed within a pair of curly braces.

→ The body of the structure should end with semi-colon.

→ Members of structures are not variables. They don't occupy any memory until they are associated with structure variables.

→ Each member of a structure contains its own memory.

Example1:-

⇒ Let us consider, we want to represent a book data-base.

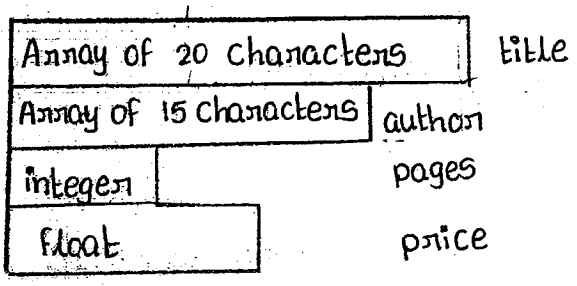
The book database consists of bookname, author, number of pages, and price. We can define a structure to represent this as follows:

```
struct book_bank
{
    char title[20];
    char author[30];
    int pages;
    float price;
};
```

→ The keyword struct declares a structure to hold the details of 4 data fields namely title, author, pages and price. These fields are called structure members (or) structure elements.

→ book_bank is the name of the structure.

→ The structure definition describes a format called template to represent the following information:



Example2:-

⇒ Define a structure to represent student information.

The student information may include roll number, student name, branch and semester.

```
struct student
{
    int roll_no;
    char name[20];
    int semester;
    char branch[10];
};
```

Example3:-

⇒ Define a structure to hold employee information (name, age, qualification and salary).

```
struct employee
{
    char name[30];
    int age;
    char qualification[10];
    float salary;
};
```

3. DECLARING STRUCTURE VARIABLES:-

⇒ Structure is a user-defined data type.

⇒ We can declare the structure variables using structure name any where in the program.

Syntax: `Struct structure_name Variable1, variable2, ---- variablen;`

Example:

```
Struct book_bank      →/* Structure definition */
{
    Char title[20];
    Char author[15];
    int pages;
    float price;
};
Struct book_bank b1, b2, b3; /* Declaration of structure variables */
```

Here b1, b2, b3 are structure variables of type book_bank. Each variable will have 4 fields namely title, author, pages and price.

After declaring the structure variables, memory is allocated.

⇒ It is also possible to combine both the structure definition and variables declaration in one statement as follows:

```
Struct book_bank
{
    Char title[20];
    Char authors[15];
    int pages;
    float price;
} b1, b2, b3;
```

⇒ The use of structure name (or) tag name is optional

Ex:

```
Struct
{
    Char title[20];
    Char author[15];
    int pages;
} b1, b2, b3;
```

NOTE: This method is not recommended because without a tag name, we can't use it for future declaration

ACCESSING STRUCTURE MEMBERS:- (OR) USE OF DOT OPERATOR;

⇨ Structure members themself are not variables. They should be linked to the structure variables in order to make them meaningful members.

→ The link between the member and a variable is established using the . (dot) operator.

→ dot operator (.) is also known as member operator (or) period operator.

⇒ We can access structure members by using . dot operator.

Syntax: `Structure variable . structure member`

Ex:
```
Struck book_bank
{
    Char title[40];
    Char author[30];
    float price;
    int year;
};
Struct book_bank b1,b2;
```
⇒ structure members

⇒ Structure variables

Now we can access the structure members as follows:

i.e. b1.title → Represents title of book1

b1.author → Represents author of book1

b1.price

b1.year

b2.title → Represents title of book2

b2.author

b2.price

b2.year

STRUCTURE INITIALIZATION:-

⇒There are many ways to initialize the structure members.

1. Structure variables can be initialized at compile-time.

```
struct book_bank
{
    char title[40];
    char author[30];
    float price;
    int year;
} b1 = {"CDS", "Balaguruswamy", 300.00, 1987};
```

This initialization assign CDS to b1.title
Balaguruswamy to b1.author
300.00 to b1.price
1987 to b1.year

NOTE:- The order of values enclosed in the braces must match the order of members in the structure definition.

Ex: We can also declare more than one structure variable and initialize them.

```
struct book_bank
{
    char title[40];
    char author[30];
    float price;
    int year;
};
struct book_bank b1 = {"CDS", "Balaguruswamy", 300.00, 1987};
struct book_bank b2 = {"SE", "Pressman", 450.00, 1990};
struct book_bank b3 = {"Networks", "Pressman", 500.00 1986};
```

2. We can also use assignment operator to initialize structure variables.

```
Ex: struct book_bank b1;
    b1.price = 300.00          strcpy(b1.author, "Balaguruswamy")
    b1.year = 1987
    strcpy(b1.title, "CDS")
```

3. Scanf function can be used to read the values through keyboard.

Ex:
```
struct book_bank
{
    char title[30];
    char author[40];
    float price;
    int year;
};
struct book_bank b;
```
⇒ `scanf("%s", b.title);`
`scanf("%s", b.author);`
`scanf("%f", &b.price);`
`scanf("%d", &b.year);`

NOTE:-

⇨ We cann't initialize individual members inside structure definition.

Ex:
```
struct book_bank
{
    float price = 300.00;
    char title[30] = "COS";
};
```
⇒ Invalid

⇨ It is permitted to have a partial initialization. we can initialize only the first few members and leave the remaining blank.

⇨ The uninitialized members will be assigned default values.

→ 0 for integer & floating-point members

→ '\0' for characters and strings.

EXAMPLE PROGRAMS:-

⇨ Define a structure book bank which contains title, author, price and year of publication. Write a c program to read this information from the keyboard and display the same on the screen.

```
#include<stdio.h>
#include<conio.h>
Void main()
{
    Struct book_bank   /* Defininig a structure */
    {
        Char title[40];
        Char author[30];
        Float price;
        Int year;
    };
    Struct book_bank b; /* Declaring a structure variable */
    Clrscr();
    Printf("Enter title of the book:");
    Scanf("%s", b.title);
    Printf(" Enter author name:");
    Scanf("%s", b.author);
    Printf("Enter price of the book:");
    Scanf("%f", &b.price);
    Printf(" Enter year of publication:");
    Scanf("%d", &b.year);
    Printf(" Book details are:");
    Printf(" Book name : %s", b.title);
    Printf("\n Author : %s", b.author);
    Printf("\n Price : %f", b.price);
    Printf("\n Year : %d", b.year);
    getch();
}
```

Explanation:-

The above program defines a structure book_bank that contains 4 members namely title, author, price and year. It reads the values through the keyboard and display the information using printf function.

Example Program 3:-

⇒ Define a structure Employee which contains employee id, name, qualification and salary. Write a C program to read the values through the keyboard and display the content of the structure.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    struct Employee  /* Defining a structure */
    {
        int id;
        char name[10];
        char qualification[20];
        float salary;
    };
    struct Employee e;  /* Declaring a structure variable */
    clrscr();
    /* Reading the values for structure members */
    printf("Enter employee details[id, name, qualification, salary]:");
    scanf("%d %s %s %f",&e.id,&e.name,&e.qualification, &e.salary);
    /* Displaying the values */
    printf("Employee ~~num~~ id : %d", e.id);
    printf("\n Employee name : %s", e.name);
    printf("\n Qualification : %s", e.qualification);
    printf("\n Salary : %f", e.salary);
    getch();
}
```

INPUT:-

Enter employee details[id, name, qualification, salary]: 12 Ajit managen 5000

OUTPUT:-

Employee id : 12
Employee name : Ajit
Qualification : Managen
Salary : 50000

Example Program3:-

⇨ Write a program to print the marks obtained by two students using structures

```
#include<stdio.h>
#include<conio.h>
Void main()
{
    Struct student
    {
       int marks1, marks2, marks3;
    };
    Struct student s1 ={90, 80, 70}; /* Compile time initialization */
    Struct student s2 ={95,80, 75};
    clrscr();
    printf(" marks obtained by st student: %d %d %d", s1.marks1, s1.marks2,
                                                        s1.marks3);
    printf("\n Marks obtained by 2nd student: %d %d %d", s2.marks1, s2.marks2,
                                                        s2.marks3);
    getch();
}
```

OUTPUT:-

Marks obtained by st student : 90 80 70
Marks obtained by 2nd student : 95 80 75

SIZE OF STRUCTURES:-

⇨ Structure is a collection of different data items.

⇨ **Size of structure = sum of size of all its members**

Ex:
```
Struct student
{
    int roll_no;
    char name[20];
    char branch[10];
    float marks;
};
```

Size of student structure = sizeof(roll_no) + sizeof(name) + sizeof(branch) + sizeof(marks)

= 2 + 20 + 10 + 4

= 36 bytes.

⇨ We can use sizeof operator to know the size of a structure.

⇨ There are two ways:

1. Syntax: `sizeof(struct structure_name);`

   Ex: sizeof(struct student);

   The above statement returns size of student structure.

2. Syntax: `sizeof(structure_variable);`

   Ex: struct student s;

   sizeof(s);
   ↳ structure variable.

Example program:-

⇨ Define a structure book which contains title, author, price & year of publication. Write a C program to display the size of the structure.

```
#include<stdio.h>
#include<conio.h>
void main()
{
   struct book
   {
      char title[20];
      char author[20];
      float price;
      int year;
   };
   struct book b;
   clrscr();
   printf("Size of structure book = %d", sizeof(struct book));
   printf("\n Size of structure book = %d", sizeof(b));
   getch();
}
```

OUTPUT:-

```
Size of structure book = 46
Size of structure book = 46
```

COPYING AND COMPARING STRUCTURE VARIABLES:-

1. COPYING STRUCTURE VARIABLES:-

⇨ Two variables of same structure type can be copied using '=' operator i.e. we can copy the content of one structure variable to another structure variable.

⇨ If S1 and S2 are two structure variables of same type then the following statements are valid: S1=S2 (OR) S2=S1.

Ex:
```
struct student
{
    int id;
    char name[50];
    float marks;
};
struct student S1={135, "RAM", 90.00};
struct student S2;
```

→ The statement S2=S1 copies the content of S1 to S2.

∴ Now S2.id = 135
S2.name = RAM
S2.marks = 90.00

2. COMPARING STRUCTURE VARIABLES:-

⇨ It is not possible to compare two structure variables directly.

⇨ i.e. if S1 & S2 are two structure variables then S1==S2, S1!=S2 are not valid.

⇨ If we want to compare the structure variables, we may compare the individual members.

Ex:
```
S1.id == S2.id
strcmp(S1.name, S2.name)
S1.marks == S2.marks
```

NOTE: C doesn't permit any logical operations on structure variables.

Example Program:-

⇨ Write a C program to illustrate copying and comparision of structure variables.

```
#include<stdio.h>
#include<conio.h>
void main()
{
   struct student
   {
      char name[50];
      int id;
      float marks;
   };
   struct student s1={"DWARAK", 12, 80.00};
   struct student s2;
   clrscr();
   s2 = s1; /* copying */
   printf("Student2 name is: %s", s2.name);
   printf("\n Student2 id is: %d", s2.id);
   printf("\n Student2 marks: %f", s2.marks);
   if(s1.id==s2.id && s1.marks==s2.marks) /* Comparing */
   {
      printf("\n Both students are same");
   }
   else
   {
      printf("\n Students are not same");
   }
   getch();
}
```

OUTPUT:- Student2 name is: DWARAK
Student2 id is: 12
Student2 marks: 80.00
Both students are same

OPERATIONS ON INDIVIDUAL MEMBERS:-

⇨ A member with • operator along with its structure variable can be treated like a normal variable.

⇨ So we perform all the arithmetic operations on individual structure members.

Ex: Struct marks
```
{
  int m1;
  int m2;
  int m3;
};
struct marks s={90,80,75};
total = s.m1+s.m2+s.m3;
```

⇨ We can also apply increment and decrement operators to numeric type (i.e. integer type) members.

Ex:
```
s.m1++
++s.m1
--s.m1
```

⇨ We can also apply all relational operators to compare individual structure members.

Ex: If s1 & s2 are two structure variables of same type, then the following operations are valid

1. s1.age > s2.age
2. s1.m1 == s2.m1
3. s1.total != s2.total

⇨ We can also use individual structure members in expressions.

Ex:
```
Struct marks
{
  int m1;
  int m2;
  int m3;
};
struct marks s={90, 70, 45};
s.m3 = s.m3*10;
avg = (s.m1+s.m2+s.m3)/3;
```

Example program:-

⇒Write a program to read the marks obtained by a student in 3 subjects and calculates its sum and average.

```
#include<stdio.h>
#include<conio.h>
void main()
{
  struct marks
  {
    int m1;
    int m2;
    int m3;
  };
  struct marks s;
  int sum;
  float avg;
  clrscr();
  printf("Enter marks obtained by a student in 3 subjects:");
  scanf("%d %d %d",&s.m1,&s.m2,&s.m3);
  sum=s.m1+s.m2+s.m3;
  avg=sum/3;
  printf("\n Sum is: %d", sum);
  printf("\n Average is: %f", avg);
  getch();
}
```

OUTPUT:-

```
Enter marks obtained by a student in 3 subjects: 70 80 90
Sum is: 240
Average is: 80.00
```

ARRAY OF STRUCTURES:-

⇒ Array is a collection of similar data items. In the same way we can also define array of structures.

⇒ Whenever the same structure is to be applied to a group of people or items, in such situations array of structures will be defined.

⇒ In array of structures each element is of structure type.

⇒ For example, if we want to maintain information of 60 students in a class, we would be required to use 60 different structure variables from student 1 to student 60, which is not convenient.

A better approach would be to use an array of structures.

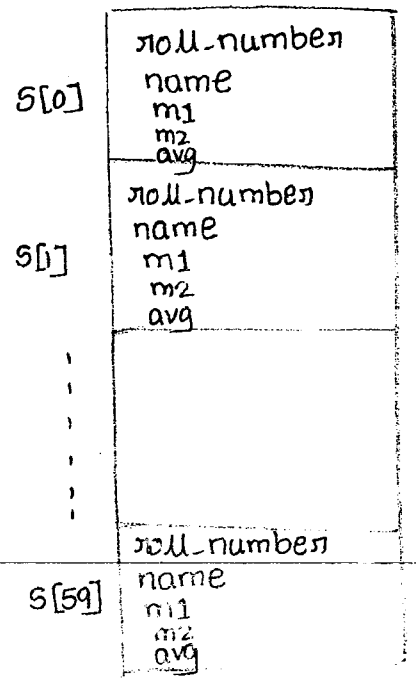
⇒ Array of structures can be declared as follows:

```
Struct student_info
{
    int roll_number;
    Char name[10];
    int m1,m2;
    float avg;
};
Struct student_info s[60];
```

→ array of structures.

In the above example, s[60] is an array of structures each containing 5 members namely roll number, name, m1, m2 and avg.

⇒ An array of structures is stored inside the memory as follows:



⇒We can use the usual array accessing methods to access individual elements and then member operator to access members.

For example, s[0].name represents first student name

s[4].avg represents 42nd student avg marks and so on..

Example program:-

```
/* Write a C-program to accept roll number, name and marks obtained
   in 3 subjects of 3 students in a class and display roll-number,
   name, marks in 3 subjects and average marks */
#include<stdio.h>
#include<conio.h>
void main()
{
    struct student
    {
       int roll_no;
       char name[30];
       int sub1, sub2, sub3;
       float avg;
    };
    struct student s[3]; /* array of structures */
    int i;
    clrscr();
    printf("Enter roll number, name, marks in 3 subjects for 3 stude");
    for(i=0;i<3;i++)
    {
       scanf("%d %s %d %d %d", &s[i].roll_no, &s[i].name,
                               &s[i].sub1, &s[i].sub2, &s[i].sub3);
    }
    /* Calculating average marks */
    for(i=0;i<3;i++)
    {
       s[i].avg = (s[i].sub1 + s[i].sub2+ s[i].sub3)/3;
    }
```

```
printf("\n Roll Number \t Name \t sub1 \t sub2 \t sub3 \t avg");
 for(i=0; i<3; i++)
 {
   printf("%d %s %d %d %d %f", s[i].roll_no, s[i].name, s[i].sub1,
                                s[i].sub2, s[i].sub3, s[i].avg );
 }
 getch();
}
```

Input:

Enter Roll number, name, marks in 3 subjects for 3 students

111	Vinod kumar	90	80	95
121	Usha	80	70	80
131	Uma	90	95	85

Output:

Roll Number	Name	sub1	sub2	sub3	avg
111	Vinod kumar	90	80	95	88.33
121	Usha	80	70	80	76.66
131	Uma	90	95	85	90.00

Explanation:-

In the above program array of structures is used to store the details of 3 students in a class.

ARRAYS WITHIN STRUCTURES:-

⇒ C allows arrays as structure members. i.e. we can use single-dimensional or multi-dimensional arrays of type int, float or char.

Ex:
```
struct student
{
    int number;
    char name[30];
    float subject[6];
};
```
→ arrays within structures

Here, the member subject contains 6 elements, subject[0], subject[1], subject[3], subject[4], subject[5]. These elements can be accessed by using appropriate subscripts.

For example S.subject[1] → represents marks obtained in 2nd subject

S.subject[5] → represents marks obtained in 6th subject

Example program:-

/* Define a structure student which contains Roll-number and marks obtained by a student in 3 subjects. Use array to represent marks. Calculate and display total marks */

```
#include<stdio.h>
#include<conio.h>
void main()
{
   struct student
   {
      int roll_no;
      int sub[3];
      int total;
   };
   struct student S;
   int i;
   printf("Enter Student Roll-Number:");
   scanf("%d", &S.roll_no);
   printf("Marks obtained in 3 subjects:");
   for(i=0; i<3; i++)
   scanf("%d", &S.sub[i]);
   S.total=0;
   for(i=0; i<3; i++)
   {
      S.total=S.total+S.sub[i];
   }
   printf("Total marks=%d", S.total);
   getch();
}
```

STRUCTURES WITHIN STRUCTURES:-

⇒ Structures within structures means nesting of structures.

⇒ Nesting of structures is allowed in C.

⇒ For example consider the following structure to store a ~~person~~ employee details.

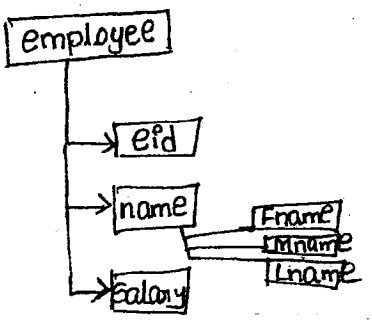
```
Struct employee
{
    int eid;
    Char Fname[20];
    Char Mname[30];
    Char Lname[20];
    float salary;
}e;
```

The above structure defines employee id, First name, middle name, Last name and salary. We can group all the members related to name together and declare them as a sub-structure as follows:

```
Struct employee
{
    int eid;
    Struct ename
    {
        Char Fname[20];
        Char Mname[30];
        Char Lname[20];
    }name;
    float salary;
}e;
```

Outer Structure

Substructure (or) Inner-Structure.



Here the structure employee contains name as its member, which itself is a structure with 3 members.

⇒ Accessing inner structure members:

outer-structure variable . inner-structure variable . inner structure member

⇒ The inner structure members can be accessed as follows:

```
e.name.Fname
e.name.Mname
e.name.Lname
```

⇒ An inner structure can have more than one variable.

⇒ We can also use tagname (or) structure name to declare inner structure.

Ex:
```
struct ename
{
    char Fname[20];
    char Mname[30];
    char Lname[30];
};
struct employee
{
    int eid;
    struct ename name; → /* we are using structure name to declare inner structure */
    float salary;
};
```

⇒ It is also possible to nest more than one structure.

Ex:
```
struct employee
{
    struct ename name;
    struct date dob;
    struct address address_part;
    ----------
    ----------
};
```
⇒ sub structures (or) inner structures.

Note: C permits nesting upto 15 levels.

C99 allows 63 levels of nesting

Example program:-

```
/* Write a program to enter full name and date of birth of an employee
   person and display the same. Use nested structures */
#include<stdio.h>
#include<conio.h>
void main()
{
   struct employee
   {
      struct ename
      {
         char Fname[20];
         char Mname[30];
         char Lname[30];
      }name;
      struct dob
      {
         int date;
         int month;
         int year;
      }d;
   };
   struct employee e;
   clrscr();
   printf("Enter name (First name, middle name and Last name):");
   scanf("%s %s %s",&e.name.Fname,&e.name.Mname,&e.name.Lname);
   printf("Enter Date of birth (Date, month, year):");
   scanf("%d %d %d",&e.d.date,&e.d.month,&e.d.year);
   printf("Name: %s %s %s",e.name.Fname,e.name.Mname,e.name.Lname);
   printf("\n Date of Birth: %d %d %d",e.d.date,e.d.month,e.d.year);
   getch();
}
```

employee Full name

Date of birth

Explanation:-

In the above program employee name and date of birth i.e dob are declared as sub structures.

Input:-

Enter name (First name, middle name, Last name): Ram Sham Pande

Enter date of birth (Date, month, year): 1 1 1989.

output:-

Name: Ram Sham Pande

Date of Birth: 1 1 1989

DIFFERENCE BETWEEN STRUCTURE and ARRAY:-

ARRAY

1. Array is a collection of similar data items

Ex:
```
int a[100];
float m[10];
```

2. Arrays can be used to represent list of similar data items.

3. To copy elements of one array to another array of same datatype, elements are copied one by one. It is not possible to copy all elements at a time.

Ex:
```
int a[3]={1,2,3};
int b[3];
b[0]=a[0]
b[1]=a[1]
b[2]=a[2]
```
} elements are copied one by one.

4. Uses of arrays:
To represent
→ List of employees in a company
→ List of products
→ List of student marks etc

STRUCTURE

1. Structure is a collection of different data items.

Ex:
```
Struct student
{
    int number;
    Char name[20];
    float marks;
};
```

2. Structures can be used to handle mixed data items (or) complex data

3. In structures, it is possible to copy the entire content of one structure variable to another structure variable of same type at a time using '=' operator

Ex:
```
Struct student
{
    int number;
    char name[20];
    float marks
};
Struct student S1={1,"usha",90.00};
Struct student S2;
S2=S1; → copying entire structure at a time.
```

4. Structures can be used to represent:
→ Customer details: name, phone number, city
→ Address : door no, street, city
→ Book details : title, author, price etc.

POINTER TO STRUCTURES:-

Pointer is a variable, which stores the address of another variable. The variable may be of any data type i.e. int, float, double etc.

In the same way we can also define a pointer to structure:

We can have a pointer pointing to a structure. Such pointers are known as "structure pointers".

Ex:
```
struct book
{
   char name[30];
   char author[20];
   int pages;
};
struct book *ptr; → ptr is a pointer to structure book.
struct book b; ⟶ structure variable.
```

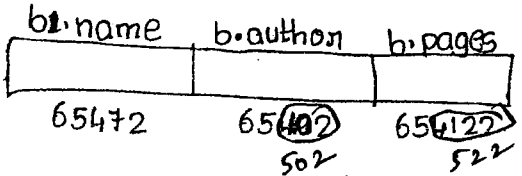
⇒ Address of structure variable can be ~~use~~ assigned to structure pointer as follows:

```
ptr = &b;
```

⇒ Now we can use "→" operator to access structure members using pointer.

Ex:
```
ptr->name        (*ptr).name
ptr->author  (OR) (*ptr).author
ptr->pages       (*ptr).pages
```

⇒ The arrangement of structure variable and pointer to structure is shown below:



ptr = &b ⇒ ptr: 65472 → address of structure member.

65524

Example program:-

/* Write a program to define a structure book. Display book details using pointer to structure */

```
#include<stdio.h>
#include<conio.h>
Void main()
{
    Struct book
    {
       Char name[25];
       Char author[30];
       Int pages;
    };
    Struct book b={"CDS", "Balaguruswamy", 300};
    Struct book *ptr;
    Clrscr();
    ptr = &b;
    Printf("Book name : %s", ptr->name);
    Printf("\n Author : %s", ptr->author);
    Printf("\n Pages : %d", ptr->pages);
    getch();
}
```

Output: Book name: CDS
Author: Balaguruswamy
Pages: 300

STRUCTURES AND FUNCTIONS:-

⇨ Like any other variables, structure variables can also be passed as arguments to functions

⇨ There are 3 methods to pass structure members as arguments to functions.

1. Passing each member of the structure as an actual arguments of the function call.
2. Passing a copy of entire structure to the called function.
3. Passing address of entire structure to the called function.

First method : Passing each member of the structure as an actual argume

⇨ We can pass each member of the structure as an actual argument to the function call.

→ The actual arguments are then treated like ordinary variables.

Syntax: functionname(structure_members);

Ex:
```
struct book
{
    char title[10];
    char author[10];
    float price;
};
struct book b={"COS","Balaguru",30.000};
display(b.title,b.author,b.price);
```
↳ Here we are passing each member of structure as an argument.

Disadvantage:-

⇨ This method is inefficient when the structure size is large.

Example Program:-

⇨ Write a program to pass structure members as arguments to a user-defined function and display the content of structure.

```
#include<stdio.h>
#include<conio.h>
Struct book
{
    char name[10];
    char author[20];
    float price;
    int pages;
};
Void main()
{
   Struct book b={"CDS","Balaguruswamy",300.00,250};
   Void display(char t[10],char a[20], float p, int pages);
   clrscr();
   printf("Book details are:");
   display(b.name, b.author, b.price, b.pages); →/* we are passing each
                                                    member as argument */
   getch();
}
void display(char t[10], char a[20], float p, int pages)
{
    printf("\n Book name is: %s", t);
    printf("\n Author is : %s", a);
    printf("\n Price : %f", p);
    printf("\n Number of pages : %d", pages);
}
```

Explanation:-

In the above example, the structure book contains 4 members namely name, author, price and pages. we are defining one structure variable b. The function display is called by passing each member of the structure as arguments i.e. display(b.name, b.author, b.price, b.pages);

SECOND METHOD: Passing entire structure to the function

⇨ We can pass a copy of entire structure to the called function.

⇨ Since the function is working on a copy of the structure, any changes made to structure members within the function will not reflect the original structure.

Syntax: function_name(structure_variable);

⇨ The function definition should take the following form:

```
return_type function_name(struct structure_name structure_variable)
{
    statement 1;
    statement 2;
    -------
    -------
    return (expression);
}
```

⇨ The function declaration is similar to function header.

⇨ The structure variable used in the function call and in the function definition must be of same struct type.

⇨ The return statement is necessary only when the function is returning some data back to the calling function.

⇨ When a function returns a structure, it must be assigned to structure of identical type in the calling function.

Ex:
```
struct book
{
    char title[20];
    char author[30];
    float price;
    int pages;
};
struct book b={"CDS", "Balaguruswamy", 300.00};
```

→ We can pass above structure to a user-defined function as follows:

```
display(b);
```

→ We are passing entire structure.

Example Program:-

⇨ Write a program to pass entire structure to a user-defined function.

```
/* Passing entire structure to the function */
#include<stdio.h>
#include<conio.h>
struct book
{
    char title[10];
    char author[20];
    float price;
    int pages;
};
void main()
{
   struct book b={"CDS", "Balaguruswamy", 300.00, 250};
   void display(struct book b1); /* Function Declaration */
   clrscr();
   printf("Book details are:");
   display(b); --> /* Passing structure variable */
   getch();
}
void display(struct book b1)  --> /* Function definition */
{
    printf("\n Book name is: %s", b1.title);
    printf("\n Author is: %s", b1.author);
    printf("\n Price : %f", b1.price);
    printf("\n Number of pages: %d", b1.pages);
}
```

THIRD METHOD: Passing address of a structure as an argument

⇨ Third method is to pass address location of the structure as an argument to the called function.

⇨ When we are passing address of a structure as an argument, the receiving parameter must be a pointer to the structure. The function can access the structure members using "->" operator.

⇨ Any changes made to the structure members within the function will reflect the original array.

⇨ This method is more efficient as compared to the second method.

Syntax: `function_name(&structure_variable);`

Example program:-

⇨ Write a program to pass address of a structure as an argument to a user-defined function and display the contents of the structure.

```
#include<stdio.h>
struct book
{
    char title[10];
    char author[20];
    float price;
    int pages;
};
void main()
{
   struct book b={"CDS", "Balaguruswamy", 300.00, 250};
   void display(struct book *b1);
   clrscr();
   display(&b); → /* passing address of a structure variable */
   getch();
}
void display(struct book *b1)  ——→ /* The receiving parameter must be a pointer */
{
    printf("Book name is: %s", b1->title);
    printf("Author is: %s", b1->author);
    printf("Price: %f", b1->price);
    printf("Number of pages: %d", b1->pages);
}
```

function returning structure variable:

Ex:

```
struct product
{
    char name[20];
    int no;
};
struct product items;
struct product fun(void);

main()
{
    clrscr();
    items = fun();
    printf("-product Name is %s\n", items.name);
    printf("No. & product is %d", items.no);
    getch();
}
struct product fun()
{
    struct product items1;
    printf("Enter product name:");
    gets(items1.name);
    printf("Enter no. & products:");
    scanf("%d", &items1.no);
    return(items1);
}
```

UNIONS:-

Union is a collection of heterogeneous elements. ie. it is a collection of different data items.

⇒ Unions are similar to structures except in terms of storage.

⇒ In structure each member has its own memory location, whereas in unions all the members use same memory location.

⇒ Unions may contain many members, but it can handle only one member at a time.

Defining & declaring unions:-

⇒ Unions can be declared using the keyword union as follows:

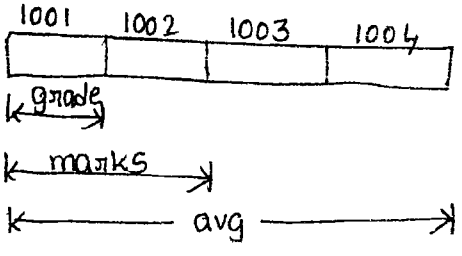
Syntax:
```
union union_name
{
    datatype member1;
    datatype member2;
    - - - - - - - - - -
    - - - - - - - - - -
};
```

Ex:
```
union results
{
    int marks;
    char grade;
    float avg;
};
```

⇒ Size of union is equal to the size of its largest member.

In the above example number of bytes reserved in the memory for the union results is 4 bytes [i.e. size of its largest member float].

⇒ All the union members shares the same memory location.



⇒ Unions can access only one member at a time because there is only one memory location.

⇒ Accessing union members is equal to accessing structure members.

Ex: r.marks,
r.grade
r.avg

⇒ Unions may be used in all case where the structure is allowed.

⇒ During accessing we should make sure that we are accessing the member whose value is currently stored in the memory.

Example program:-

⇒ Write a program that uses sizeof operator to differentiate between structure and union.

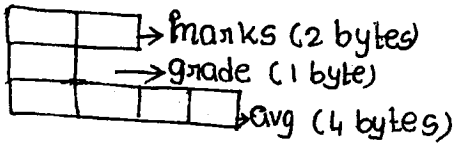
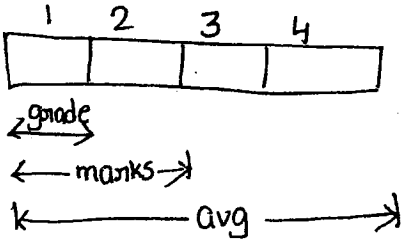
```
#include<stdio.h>
void main()
{
    union result
    {
        int marks;
        char grade;
        float avg;
    }r1;
    struct res
    {
        int marks;
        char grade;
        float avg;
    }r2;
    clrscr();
    printf("Size of union: %d bytes", sizeof(r1));
    printf("\n Size of structure: %d bytes", sizeof(r2));
    getch();
}
```

Output: Size of union: 4 bytes
Size of structure: 7 bytes

DIFFERENCES BETWEEN STRUCTURE and UNION:-

Structure	Union.

1. To define a structure the keyword 'struct' is used.
Ex: struct result
{
int marks;
char grade;
float avg;
};
	1. To define a union, the keyword union is used.
	Ex: union result
	{
	int marks;
	char grade;
	float avg;
	}u;

2. In structure each member has its own memory location 	2. In unions, all the members shares the same memory location 
3. The size of structure is equal to the sum of size of all its members. Ex: sizeof(result) = Sizeof(marks) + Sizeof(grade) + Sizeof(avg) = 2+1+4 = 7 bytes	3. The size of union is equal to the size of its largest member Ex: sizeof(result) = sizeof(avg) = 4 bytes
4. We can access all structure members at a time.	4. In unions, we can access only one member at a time.

BIT FIELDS:-

We know that integer numbers requires 16 bits to store data. There are some situations, where data items requires less than 16 bits space. In such cases we waste the memory space.

~~To overcome this C-supports a concept known as "bit-fields".~~

⇒ Using bitfields we can tell the compiler to allocate only specified number of bits.

Bit field:- A bit field is a set of adjacent bits whose size can be from 1 to 16 bits in length.

⇒ The name and sizeof bitfields can be defined using a structure.

Defining bitfields:-

Syntax:
```
struct tag_name
{
   data-type name1 : bit_length;
   data-type name2 : bit_length;
   - - - - - - - - - - - -
   - - - - - - - - - - - -
   data-type nameN : bit_length;
};
```

where data-type is either int (or) unsigned int (or) signed int.

bit_length is the number of bits used for specified name.

⇒ The bit_length is decided by the range of value to be stored. The largest value that can be stored is (2^{n-1}), where n is bit_length.

Ex:
```
struct personal
{
   unsigned age:3
   int result:1
   unsigned count:4
} p;
```

⇒ The range of values is $2^n - 1$, as follows:

Bit field	Bit_length	Range of values
age	3	0 to 2^3-1
result	1	0 to 2^1-1
count	4	0 to 2^4-1

2^4-1

The above example defines a variable p with 3 bit fields.

⇒ Once bit fields are defined, they can be accessed using '.' operator.

Ex: p.age, p.result, p.count.

⇒ We can't use scanf to read values into a bitfield.

⇒ We can use assignment operator to assign values to the bitfields

Ex: p.age = 17
p.result = 1

⇒ Bitfields can be used in expressions

Ex: p.age = p.age * 2
if(p.age == 18)

⇒ Bitfields can't be arrays.

⇒ We can't take address of a bitfield variable.

⇒ We can't use pointer to access the bit fields.

⇒ There can be unnamed bitfields with declared size.

Ex: unsigned : bit_length.

⇒ It is possible to combine normal structure elements with bitfield elements.

Ex:
```
struct personal
{
    char name[10];
    float salary;          } → Normal structure members
    int age : 3;
    int result : 1;
    unsigned count : 4;    } → Bit-fields.
};
```

Ex: Write a C program to display the examination result of student using bit field.

```
#define pass 1
#define fail 0
#define A 0
#define B 2
#define C 3

main()
{
    struct stud
    {
        char name[20];
        unsigned result : 1;
        unsigned grade : 2;
    } s;

    strcpy(s.name, "rachu");
    s.result = PASS;
    s.grade = C;
    printf("Name : %s", s.name);
    printf("result : %d", s.result);
    printf("Grade : %d", s.grade);
}
```

OM VINAYAKA
OM SAI RAM

UNIT-5

FILES

→ INTRODUCTION
→ TYPES OF FILES
→ Defining and opening a file
→ Closing a file
→ Input/output operations on files
→ Error handling during I/O operations
→ Random access to files
→ Command line arguments
→ Application of command line arguments.

INTRODUCTION:-

REASONS FOR APPROACHING FILES:-

We have been using the functions such as scanf() and printf() to read and write data. These are console-oriented I/O functions which always use the terminal (keyboard & screen) as the target place.

This works fine as long as the data is small. However many real-life problems involve large volumes of data. In such situations, the console I/O functions pose two major problems:

1. It becomes 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 permanently on the disks and read whenever necessary, without destroying the data.

Files allows us to store information permanently in the disk.

FILE:- A file is a place on the disk, where a group of related data is stored.

(OR)

A file is a collection of records that can be accessed through a set of library functions.

BASIC FILE OPERATIONS:-

⇒ C-supports a number of functions to perform the basic file operations.

1. Creating a file
2. Opening a file
3. Reading data from a file
4. Writing data to a file
5. Closing a file.

⇒ There are two ways to perform file operations in C.

1. Low-level I/O : Uses UNIX System calls
2. High-level I/O : Uses functions in C's standard I/O library.

High-Level I/O Functions:-

Function name	Operation
fopen()	- Creates a new file for use. - Opens an existing file for use.
fclose()	— Closes a file which has been opened for use.
getc()	- Reads a character from a file
putc()	- Writes a character to a file
getw()	- Reads an integer from a file
putw()	- Writes an integer to a file
fprintf()	- Writes a set of data values to a file
fscanf()	- Reads a set of data values from a file
fseek()	- Sets the position to a desired position in the file.
ftell()	- Gives the current position in the file.
rewind()	- Sets the position to the beginning of the file.

2. TYPES OF FILES:-

⇒ Files are used for storage and retrieval of data by a C program.

⇒ Depending on the format in which data is stored, files are divided into two types: (i) Text files
(ii) Binary files

(i) Text files:-

→ Text file stores textual information like alphabets, digits and symbols etc.

→ Text files can be read and understood by human beings.

→ Since data is stored in memory in the binary format, the text file is first converted into the binary form before it is stored in memory.

→ A text file can store different character set such as: A to Z
a to z
1, 2, ---9
Special symbols etc.

Ex: C source code files and files with .txt extensions.

(ii) Binary files:-

→ Any file which stores the data in the form of bytes is known as binary file.

→ A binary file stores the information in binary format i.e. 0's and 1's.

→ The use of binary files eliminates the need of data conversion from text to binary format

→ The main drawback of binary file is that the data stored in the binary file is not in a human readable form.

Ex: Every executable file generated by a C-compiler is a binary file.
All files with .exe extensions
Image files etc

DEFINING and OPENING A FILE:-

⇨ A file has to be opened before beginning of read and write operations.

⇨ If we want to store data in a file, we need to specify certain things about the file.

1. Filename: A string of characters that make up a valid filename.

   Ex: input, name.txt, file.doc, add.c etc.

2. Data structure - FILE

   All files should be declared as type FILE before they are used. FILE is a defined data type.

3. Purpose: Purpose of opening a file.

Syntax:

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

Where, FILE → structure defined in I/O Library

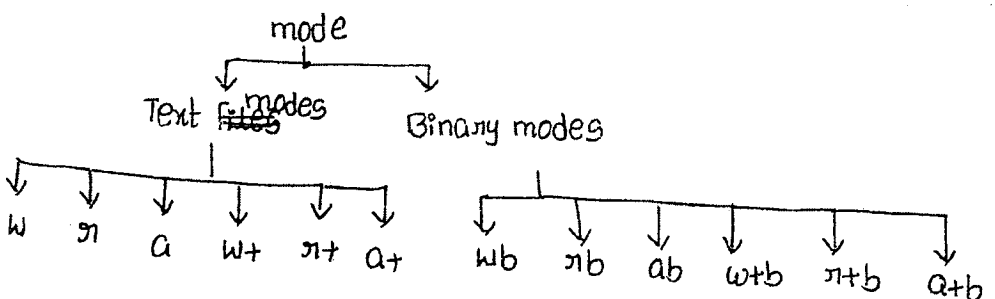
fp → is a pointer to the data type FILE

filename → valid identifier

mode → Specifies the purpose of opening the file.

⇨ fopen function opens the specified file in specified mode.

⇨ Mode can be any one of the following:



TEXT MODES:-

1. w (write): write only

→ When the mode is "w", a file with the specified file name is created if the file doesn't exist.

→ If the file already exist, it will delete the contents of the file and file is opened as a new file.

Ex:
```
FILE *fp;
fp = fopen("data.txt", "w");
```
"w" → mode; "data.txt" → Filename

2. r(read) : Opens the file for reading only

⇒ When the mode is "r", fopen function opens a pre-existing file for reading. If the file doesn't exist, then the compiler returns NULL to the file pointer.

Ex: fp=fopen("data.txt", "r");
if(fp==NULL)
printf("File doesn't exist");

3. a (append): Opens a file for appending data

⇒ When the mode is "a", fopen function opens a pre-existing file in append mode without erasing its contents and append data at the end of the file.

⇒ If the file doesn't exist the mode of "a" is same as "w".

Ex: fp = fopen("data.txt", "a");

4. w+ (write+read):-

The file is opened for both reading and writing.

Ex: fp=fopen("data.txt", "w+");

5. r+ (read+write):-

This mode opens the file for both reading and writing.

Ex: fp=fopen("data.txt", "r+");

6. a+ (append+read):-

When the mode is "a+", the file can be read and data can be appended at the end of the file.

Ex: fp = fopen("data.txt", "a+");

CLOSING A FILE:-

→ A file must be closed as soon as all operations on it have been completed.

→ Closing a file prevents any accidental misuse of the file. A file must be closed when we want to reopen the same file in different mode.

→ The function fclose is used to close a file.

Syntax: fclose(file-pointer);

Ex: FILE *fp1, *fp2;

```
fp1 = fopen("input.c", "w");
fp2 = fopen("output.c", "r");
- - - - - - - - - - - - -
- - - - - - - - - - - - -
```

fclose(fp1); → close the file associated with file pointer fp1.

fclose(fp2); → close the file associated with file pointer fp2.

⇒ Once a file is closed, its file pointer can be reused for another file.

⇒ To close more than one file at a time fcloseall() function is used.

Syntax: `fcloseall();`

INPUT/OUTPUT OPERATIONS ON FILES:-

Once a file is opened, reading (or) writing operations are accomplished using the standard I/o functions.

1. getc() and putc() functions:-

getc():- getc() function is used to read a character from a file that has been opened in read mode.

Syntax: `ch = getc(fp);` (fp → File pointer)

ch → character variable

→ getc() function reads a character from the file whose file pointer is fp and moves the file pointer to the next location immediately.

→ getc() function returns EOF, if the end of file is reached. Therefore reading should be terminated when EOF is encountered.

Ex:
```
fp = fopen("sample.txt", "r");
char ch;
while ((ch=getc(fp)) != EOF) /* Reads characters from the file
                                until EOF is reached */
printf("%c", ch);
```

2. putc() function:-

putc() function is used to write a single character to a file.

Syntax: `putc(c,fp);`

where c - is a character variable

fp - is a file pointer

→ putc() function writes the character contained in the character variable 'c' to the file associated with FILE pointer fp

→ File pointer moves by one character position for every operation of getc or putc.

NOTE:- The end-of-the file is indicated by entering an EOF character, which is Control-z in a new line.

Example program:- Write a program to read data from the keyboard, write it to a file. Again read same data from the file, and display it on the screen.

```
#include<stdio.h>
void main()
{
    FILE *fp;
    char c;
    clrscr();
    fp=fopen("sample.txt","w"); /* opening file in write mode */
    while((c=getchar())!=EOF)   /* Get a character from keyboard */
    {
        putc(c,fp);  /* Write a character to sample.txt file */
    }
    fclose(fp);
    fp=fopen("sample.txt","r"); /* opening file in read mode */
    while((c=getc(fp))!=EOF)  /* Read a character from the file */
    {
        printf("%c",c);   /* Display a character on screen */
    }
    fclose(fp);  /* closing the file sample.txt */
  getch();
}
```

putw and getw Functions:- The getw and putw are integer-oriented functions.

1. putw:- putw function is used to write an integer value to a file which is pointed by file pointer fp.

Syntax: `putw(x, fp);`

where x is a integer variable.

2. getw:- getw() function is used to read an integer value from a file.

Syntax:
```
int x;
x=getw(fp);
```

getw() function reads an integer value from a file whose file pointer is fp

⇒ getw() and putw() functions are useful when we deal with integer data.

Example program:-

⇒ Write a program to write integers to a file and again read the integers from the same file and display it on the screen.

```
#include<stdio.h>
void main()
{
   FILE *fp;
   int x;
   clrscr();
   fp=fopen("numbers.txt","w");
   printf("Enter integer numbers:");
   while(1)
   {
     scanf("%d",&x);
     if(x==0)  /* if the entered number is '0', it stops */
     break;
     else
     putw(x,fp); /* writing integer number to the file number.txt */
   }
   fclose(fp);
   fp=fopen("number.txt","r");
   while((x=getw(fp))!=EOF)  /* Reading integer values from the file */
    printf("%d",x);
   getch();
}
```

3. fprintf and fscanf functions:-

fprintf and fscanf functions are used to handle group of mixed data items simultaneously.

1. fprintf():-

fprintf() function is used for writing characters, strings, integers, floating point values etc to a file.

Syntax: `fprintf (fp, "control strings", variable_list);`

Where fp → fp is a file pointer associated with a file that has been opened for writing.

Control string → Specifies the format specifications for the variables in the variable list.

Variable_list → The variable list may include variables, constants and strings.

Example:- `fprintf(fp, "%s %d %f", name, age, 7.5);`

The above function writes a string 'name', an integer age and a floating point number 7.5 to the file associated with file pointer fp.

2. fscanf():-

fscanf() function is used to read mixed data items from a file.

Syntax: `fscanf(fp, "control-strings", list);`

The above statement reads the items in the list from the file specified by fp, according to the specifications contained in the control string.

Example: `fscanf(fp, "%s %d", &name, &quantity);`

⇒ fscanf function returns number of items successfully read.

When end of file is reached, it returns EOF

Example program:-

⇒ Write a program to write data into the text file and the same using fprintf and fscanf functions.

```
#include<stdio.h>
Void main()
{
    FILE *fp;
    Char name[20];
    Int age;
    Float height;
    fp=fopen("data.txt", "w");
    printf("Enter name, age and height:");
    Scanf("%s %d %f", &name, &age, &height);
    fprintf(fp, "%s %d %f", name, age, height); /* Writing mixed data items to a file */

    Fclose(fp);
    fp=fopen("data.txt", "r");
    fscanf (fp, "%s %d %f", &name, &age, &height); /* Reading data items from a file */
    printf("Name is: %s", name);
    printf("\n Age: %d", age);
    printf("\n Height: %f", height);
    fclose(fp);
    getch();
}
```

Input:- Enter name, age and height : puja 25 5.5

Output: Name is: puja
Age: 25
Height: 5.5

ERROR HANDLING DURING I/O OPERATIONS:-

⇒ It is possible that an error may occur during I/O operations on a file.

⇒ Typical error situations include:

1. Trying to read beyond the end-of-the file mark.
2. Device overflow.
3. Trying to use a file that has not been opened
4. Trying to perform an operation on a file, when the file is opened for another type of operation.
5. Opening a file with an invalid filename.

⇒ If we fail to check these errors, a program may behave abnormally when an error occurs. An unchecked error may result in termination of the program or incorrect output.

Therefore it is necessary to check these errors during I/O operations on a file.

C-language supports two library functions to detect I/O errors in the files.

1. feof()
2. ferror()

1. feof():-

The feof function is used to test for an end of file condition.

Syntax: `feof(fp);`

→ feof function takes FILE pointer fp as its only argument

→ feof() function returns non-zero integer value if the file pointer is at the end of the file. Otherwise it returns zero.

```
Ex: if( feof(fp) != 0)
        printf("End of data (or) file");
```

would display the message "End of data" on reaching the end of file condition.

2. ferror():-

→ This function is used for detecting any error that might occur during read/write operations on a file.

→ It takes FILE pointer as its argument.

⇒ Returns non-zero integer if an error has been detected.
Returns zero otherwise.

Syntax: `ferror(fp);`

Ex: `if(ferror(fp)!=0)`
`printf("An error has occured");`

⇒ When a file is opened using fopen(), the file pointer is returned. If the file can't be opened for some reason, then the function returns a null pointer. This can be used to test whether the file is opened or not.

Ex:
```
FILE *fp;
fp=fopen("sample.txt","r");
if(fp==NULL)
printf("File can't be opened");
```

Example program:-

⇒ Write a program to detect errors that has occured during I/O operations.

```
#include<stdio.h>
void main()
{
    FILE *fp;
    char c;
    fp=fopen("sample.txt","w");
    if(fp==NULL)
    {
      printf("Can't open a file");
      exit(0);
    }
    while(!feof(fp)) /* Checking for end of file condition */
    {
        c=getc(fp);
        if(ferror(fp)!=0) /* Checking error using ferror() */
        { printf("\n An error has occured");
          exit(0);
        }
        printf("%c",c);
    }
    fclose(fp);
}
```

RANDOM ACCESS TO FILES:-

⇒ Using Random access functions we can access only a particular part of a file.

⇒ There are 3 random access functions

1. fseek()
2. ftell()
3. rewind()

1. fseek():-

The fseek function is used to move file pointer to a desired location within the file.

Syntax: `fseek(filepointer, offset, position);`

where,

→ file pointer points to the specified file

→ offset is a number or variable of type long.

offset specifies the number of positions to be moved from the location specified by position.

offset may be positive means move in forward direction (or) it may be negative → move in backward direction.

position:- position indicates the current position of the file pointer. position may take one of the following 3 values.

value	meaning
0	- Beginning of the file
1	- Current position
2	- End of the file

Examples:-

1. fseek(fp,0,0) - Go to the beginning
2. fseek(fp,0,1) - Stay at the current position
3. fseek(fp,0,2) - Go to the end of the file.
4. fseek(fp,m,0) - fp is moved to (m+1)th byte in the file.
5. fseek(fp,m,1) - moves fp in forward direction by m bytes from the current position.
6. fseek(fp,-m,1) - Go backward by m bytes from the current position
7. fseek(fp,-m,2) - Go backward by m bytes from the end.

⇨ fseek function returns zero if the operation is successful.
returns -1 if there is any error in moving file pointer.

Example program:-

⇨ Write a program to read last n characters in a file using fseek function.

```
#include<stdio.h>
void main()
{
    FILE *fp;
    char ch;
    int n;
    clrscr();
    printf("Enter number of characters to be read:");
    scanf("%d",&n);
    fp=fopen("sample.txt","r");
    if(fp==NULL)
    {
       printf("File can't be opened");
       exit(0);
    }
     fseek(fp,-n,2);
     while((ch=getc(fp))!=EOF)
     {
        printf("%c",ch);
     }
     fclose(fp);
     getch();
}
```

Example program:-

⇒ Write a program to read the text after skipping n characters from the beginning of the file.

HINT: fseek(fp, n, 0)

2. ftell():-

⇨ ftell() function returns the current position of the file pointer in a file.

Syntax:
```
int n;
n=ftell(fp);
```

⇨ This function is useful in saving the current position of a file, which can be used later in the program.

Ex
```
fseek(fp, 2, 0)
n=ftell(fp);
```
⇨ n=2

Example program:-

⇨ Write a program to print the current position of a file pointer using ftell() function

```
#include<stdio.h>
#include<conio.h>
void main()
{
    FILE *fp;
    Clrscr();
    fp=fopen("sample.txt", "w");
    if(fp==NULL)
    {
        printf("File cann't be opened");
        exit(0);
    }
    printf("At the beginning file pointer position is: %d", ftell(fp));
    fseek(fp,5,0);
    printf("\n After moving file pointer position is: %d", ftell(fp));
    fclose(fp);
    getch();
}
```

Output:

At the beginning file pointer position is: 0
After moving file pointer position is: 5

3. rewind:-

The rewind() function resets the position of file pointer to the beginning of the file.

It takes file pointer as its argument.

Syntax: `rewind(fp);`

Ex:
```
fp=fopen("sample.txt","r");
rewind(fp);
n=ftell(fp);
```
The value of n=0 because the file pointer position has been set to to the beginning of the file by rewind.

Ex2:
```
fp=fopen("sample","w");
fseek(fp,5,0);
n=ftell(fp);
```
⇨ Now n=5 because fseek() moves fp by 5 bytes
```
rewind(fp);
n=ftell(fp)
```
⇨ Now n=0 because rewind function resets the fp position to the start of the file.

⇒ ~~write a program to copy~~

APPLICATIONS OF COMMAND LINE ARGUMENTS:-

⇒ The key application of command line arguments is run-time specification of data. i.e. the programmer can specify the required data during runtime.

⇒ For example consider a situation, where you are required to copy the contents of one file into another.

As long as the source and target files remains the same, we can specify the names of the files statically within the program code.

But, if we want the program ~~code~~ to copy the contents of any file to any other file, then we must use the concept of command line arguments.

The command line arguments allow the user to specify the filenames dynamically at run time.

⇒ We can use the concept of command line arguments whenever data is required to be specified dynamically.

Example:-

⇒ Write a program to copy the contents of one file into another file using the command line arguments.

```
/* program for copying contents of one file into another */
#include<stdio.h>
#include<conio.h>
Void main(int argc, char *argv[])
{
    FILE *fp1, *fp2;
    Char ch;
    clrscr();
    if(argc!=3)
    {
        printf("Inconnect arguments");
        exit(0);
    }
    fp1=fopen(argv[1], "r");  /* opening source file in read mode */
    fp2=fopen(argv[2], "w");  /* opening target file in write mode */
```

⇒ Write a program to read the existing filename and display its contents on the moniton.

```
#include<stdio.h>
void main(int argc, char *argv[])
{
    FILE *fp;
    char c;
    clrscr();
    if(argc!=2)
    {
      printf("Inconnect arguments");
      exit(0);
    }
    fp=fopen(argv[1], "r");
    if(fp==NULL)
    {
      printf("File cann't be opened");
      exit(0);
    }
    while((c=getc(fp))!=EOF)
    {
      printf("%c", c);
    }
    fclose(fp);
    getch();
}
```

Explanation:-

In the above program existing filename is passed from the command line argument. The file is opened in read mode.

getc() function reads a character from the file. The printf() functi.. displays it on the screen.

To Run the above program:-

1. Save the program as display.c and make it executable file.
2. Switch to command prompt and type the following:

```
C:\> display file1.c
```

Example program:-

⇒ Write a program to display the number of command line arguments and their names.

```
#include<stdio.h>
Void main(int argc, char *argv[3])
{
    int i;
    clrscr();
    Printf(" Number of arguments : %d", argc);
    Printf(" Names of arguments:");
    for(i=0; i<argc; i++)
    printf(" %s", argv[i]);
    getch();
}
```

⇒ To run the above program

1. Save and compile the program. Make it as executable file.
2. Switch to command prompt and type the following:

```
C:\> sample Hello World
```

Output:

```
Number of arguments:3
Names of arguments : sample
                     Hello
                     World.
```

Command-Line arguments:-

The parameters supplied to a program when the program is executed is known as "command-line arguments".

⇒ We know that every C program should have a main function.

→ Like other functions main function can also take arguments.

⇒ The main() can take 2 arguments: 1. argc
2. argv

Syntax:
```
main(int argc, char *argv[])
{
    -------
    -------
}
```

1. argc: argument counter

The argc counts the number of arguments passed to the program from the command line.

2. argv: argument vector

It is a pointer to an array of strings which contains the names of arguments.

→ The size of argv is equal to the value of argc.

⇒ Information contained in the commandline is passed onto the program through these arguments.

⇒ To give command line arguments, the user has to switch to command prompt.

Ex: C:\>program file1 Hai

→ The first argument is always an executable program followed by arguments.

For the above example argc = 3

argv[0] → program

argv[1] → file1

argv[2] → Hai

```
while ((ch=getc(fp1)) != EOF) /* Reads character from source file */
{
    putc(ch, fp2);          /* Writes characters to the target file */
}

fcloseall();
fp2 = fopen(argv[2], "r");
printf("After copying the content of target file is:");
while ((ch=getc(fp2)) != EOF)
{
    printf("%c", ch);
}
fclose(fp2);
getch();
}
```

⇒ Now make the above program as executable file and switch to command prompt and type the following:

```
C:\> copy source.txt target.txt
```

Explanation:-

In the above program source file name and target file name is supplied as command line arguments.

The source file is opened in read mode and target file is opened in write mode.

The program reads the characters from the source file using getc() function and copies the contents to target file using putc() function.

Output:-

After copying the content of target file is: This is an application of command line arguments.

NOTE: source.txt contains : This is an application of command line arguments.

⇒ Write a program to write integers to a text file & read it.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    FILE *fp;
    int n;
    clrscr();
    fp=fopen("input.txt","w"); /* opens the file input.c in write mode */
    while(1)
    {
      scanf("%d",&n);
      if(n==0)
      break;
      else
      putw(n,fp); /* writes an integer n to the file pointed by fp */
    }
    fclose(fp);
    fp=fopen("input.txt","r"); /* opens the file in read mode */
    while((n=getw(fp))!=EOF) /* reading integers from the file */
    {
        printf("%d",n);
    }
    fclose(fp);
   getch();
}
```

Explanation:

In the above program putw() function is used to write integer to a file and getw function is used to read an integer from a file.

⇨ Write a C program to reverse the contents of a file and copies it into a new file.

```
/* Program to reverse the contents of a file and copies it into a new file */
#include<stdio.h>
#include<conio.h>
void main()
{
   FILE *fs,*ft;
   char str[100];
   int i=0;
   clrscr();
   fs=fopen("source.txt","r"); /*opening source file in read mode */
   if(fs==NULL)
   {
      printf("Source file can't be opened");
      exit(0);
   }
   ft=fopen("target.txt","w"); /* opens the target file in write mode */
   if(ft==NULL)
   {
      printf("Target file can't be opened");
      exit(0);
   }
   while((ch=getc(fs))!=EOF) /* Reads the contents of source file
   {                            character by character and stores in 'str'
      str[i]=ch;
      i++;
   }
   for(i=strlen(str)-2;i>=0;i--)
   {
      putc(str[i],ft); /* Writing the reverse of the file contents */
   }
   fclose(fs);
   fclose(ft);
   getch();
}
```

Explanation:-

The above program reverse the contents of the source file and copies it into a target file.

The source file is opened in read mode and target file is opened in write mode.

The program reads the contents of source file character by character and stores it into the string str.

The program reverse the content of source file and write it into target file using putc() function.

Reversing 'n' characters in Command line

```
main(int argc, char *argv[])
{
  char ch, c[100];
  int n, i=0;
  FILE *fp;
  clrscr();
  fp= fopen(argv[1], "r+");
  if(fp==NULL)
  { printf("file doesn't exist");
    getche();
    exit();
  }
n= atoi(argv[2]);
printf("contents of file:");
while((ch=getc(fp))!=EOF)
{ printf("%c", ch);
  c[i]=ch;
  i++;
}
rewind(fp);
for(i=n-1; i>=0; i--)
{ putc(c[i], fp);
}
fclose(fp);
printf("Contents of file after reversing is");
fp=fopen(argv[1], "r");
while((ch=getc(fp))!=EOF)
{ printf("%c", ch);
}
fclose(fp);
getch();
```