ANNAI WOMEN'S COLLEGE
(Arts and Science)

Affiliated to Bharathidasan University-Tiruchirapalli.

TNPL Road,Punnamchathram,Karur-639136.

DEPARTMENT OF COMPUTER SCIENCE,COMPUTER APPLICATIONS & IT

Faculty Name: Ms.A.HEMAPRIYA,M.Sc,M.Phil.,

Major   : I B.Sc(INFORMATION TECHNOLOGY)

Paper Code   : 16SCCIT2

Title of Paper: Programming in C

UNIT	TOPIC	PAGENO
1	**INTRODUCTION TO C**	1-27
2	**MANAGING INPUT OUTPUT OPERATIONS**	28-52
3	**ARRAYS**	53-70
4	**STRUCTURES AND UNIONS**	71-102
5	**DYNAMIC MEMORY ALLOCATION**	103-111

PROGRAMMING IN C

Objective:

To impart basic knowledge of Programming Skills in C language.

Unit I

Introduction to C – Constants, Variables, Data types – Operator and Expressions.

Unit II

Managing Input and Output operations – Decision Making and Branching – Decision making and Looping.

Unit III

Arrays – Character Arrays and Strings – User defined Functions.

Unit IV

Structures and unions – Pointers – File management in C.

Unit V

Dynamic memory allocation – Linked lists- Preprocessors – Programming Guide lines.

Text Book:

1. Balagurusamy E .,Programming in ANSI C , Sixth Edition, McGraw-Hill, 2012

Reference Book:

1. R.S.Bichkar, Programming with C, University Press, 2012

*****

UNIT-I
INTRODUCTION TO C

History of C

      C language is developed by Mr. Dennis Ritchie in the year 1972 at bell laboratory at USA, C is a simple and structure Oriented Programming Language.

In the year 1988 C programming language standardized by ANSI (American national standard institute), that version is called ANSI-C. In the year of 2000 C programming language standardized by ISO that version is called C-99. All other programming languages were derived directly or indirectly from C programming concepts.

Overview of C

      C is a computer programming language developed in 1972 by Dennis M. Ritchie at the Bell Telephone Laboratories to develop the UNIX Operating System. C is a simple and structure oriented programming language.

      C is also called mother Language of all programming Language. It is the most widely use computer programming language, This language is used for develop system software and Operating System. All other programming languages were derived directly or indirectly from C programmingconcepts.

Dennis Ritchie

In the year 1988 'C' programming language standardized by ANSI (American national standard institute), that version is called ANSI-C. In the year of 2000 'C' programming Language standardized by 'ISO' that version is called C-99

Features of C language

- It is a robust language with rich set of built-in functions and operators that can be used to write any complexprogram.
- The C compiler combines the capabilities of an assembly language with features of a high-level language.
- Programs Written in C are efficient and fast. This is due to its variety of data type and powerful operators.
- It is many time faster thanBASIC.
- C is highly portable this means that programs once written can be run on another machines with little or nomodification.
- Another important feature of C program, is its ability to extenditself.
- A C program is basically a collection of functions that are supported by C library. We can also create our own function and add it to Clibrary.
- C language is the most widely used language in operating systems and embedded system development today.
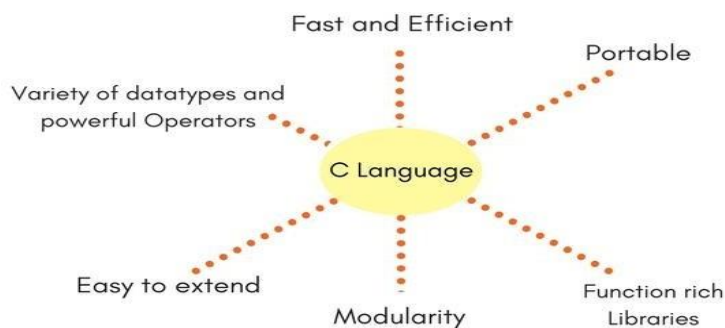
SIMPLE C PROGRAM:(Compile and Run)

Using an IDE - Turbo C

- We will recommend you to use**Turbo C** IDE, oldest IDE for c programming.
- It is freely available over internet and is good for abeginner.

Step 1 : Open turbo C IDE(Integrated Development Environment), click on **File** and then click on New
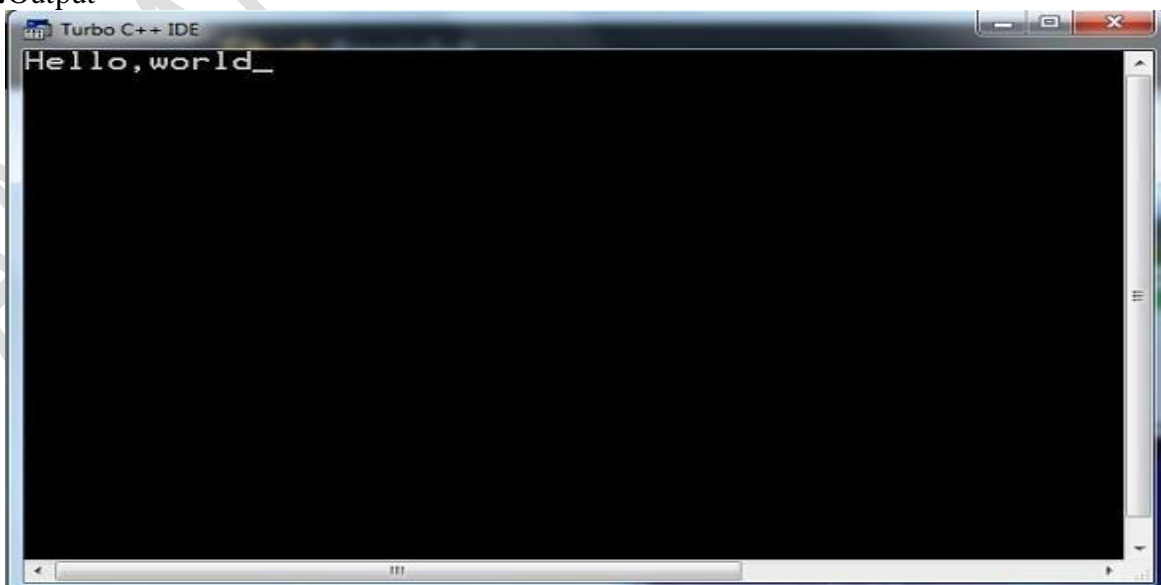

Step 2 : Write the above example as it is
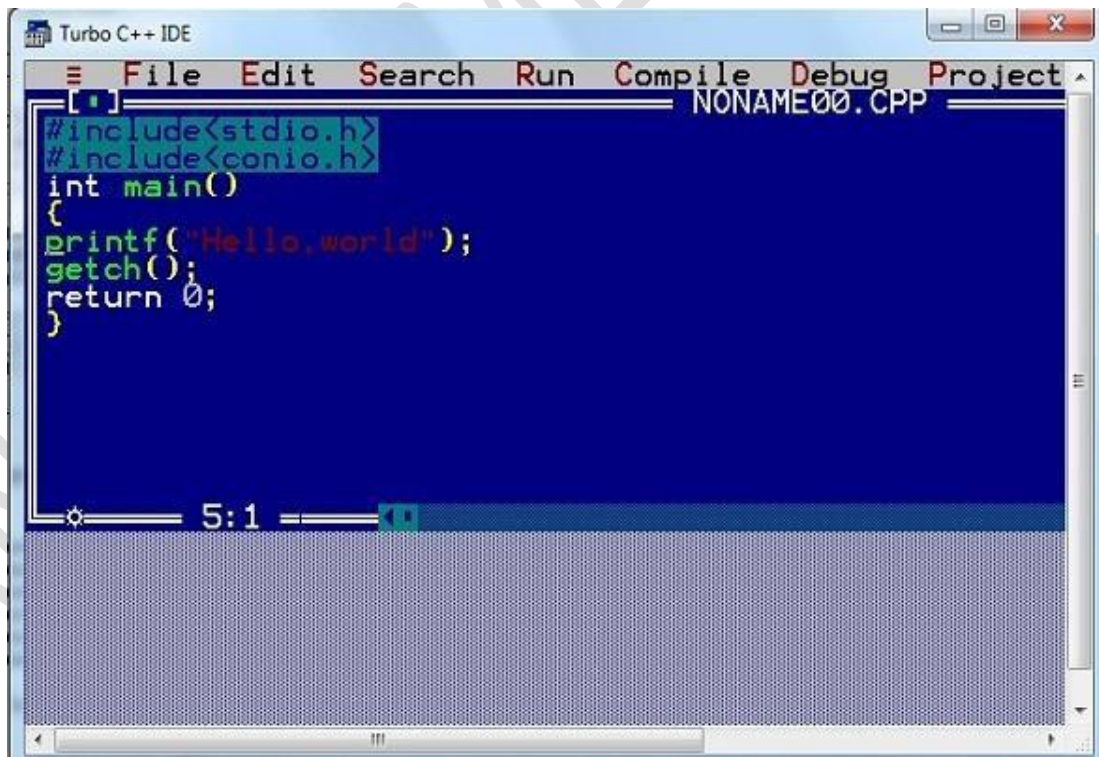
Step 3 : Click on compile or press Alt+f9 to compile the code

Step 4 : Click on Run or press Ctrl+f9 to run the code
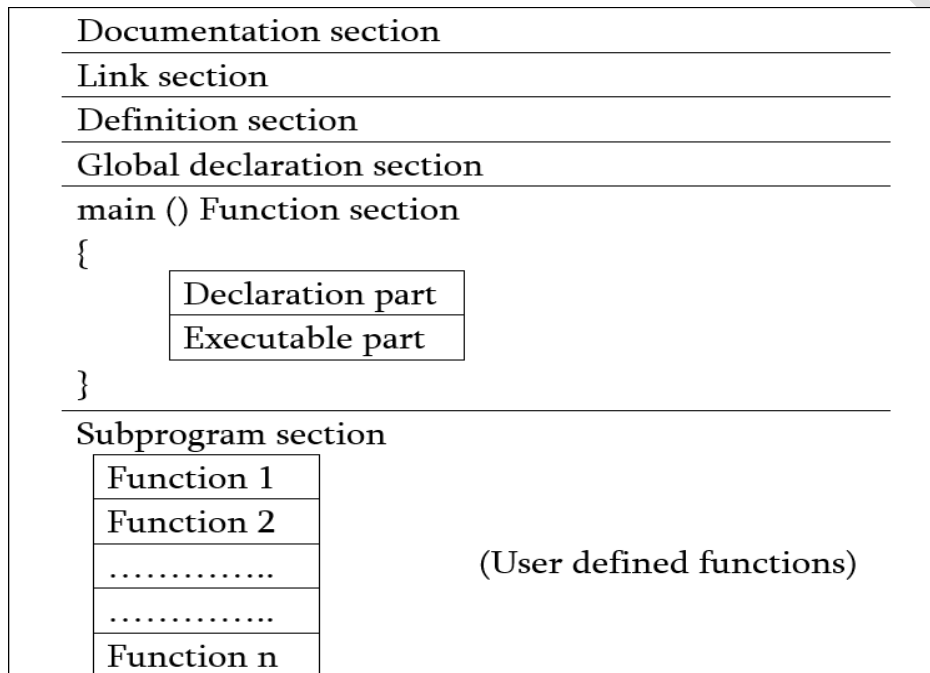

Step 5 : Output

Difference between Compile and Run

- You must be thinking why it is a 2 step process, first we compile the code and then we run the code. So, compilation is the process where the compiler checks whether the program is correct syntax wise, and there are no errors in the syntax.
- When we run a compiled program, then it actually executes the statements inside the main() function.

BASIC STRUCTURE OF C PROGRAM

- Any C program is consists of 6 main sections. Below you will find brief explanation of each of them.

Documentation section
Link section
Definition section
Global declaration section
main () Function section
{
Declaration part
Executable part
}
Subprogram section
Function 1
Function 2
…………..   (User defined functions)
…………..
Function n

Documentation Section

- This section consists of comment lines which include the name of programmer, the author and other details like time and date of writing the program.
- Documentation section helps anyone to get an overview of the program.
- There are two ways in which we can write comments.
    - ✓ **Using //** - This is use to write a single line comment.
    - ✓ **Using /* */** - The statements enclosed within /* and */ , are used to write multi-line comments.

Example of comments :

Single line comments:

```
//sum of two numbers
```

Multi-line comments:

```
/*Welcome to CProgram
You are Learning C-programming
This program is about summing of two numbers*/
```

6

Link Section

The link section consists of the header files of the functions that are used in the program. It provides instructions to the compiler to link functions from the system library.

Example:

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

Definition Section

- All the symbolic constants are written in definitionsection.
- Macros are known as symbolicconstants.

Example:

```
#define PI3.14
#define TRUE1
```

Global Declaration Section:

- There are some variables that are used in more than onefunction.

- Such variables are called global variables and are declared in the global declaration section that is outside of all thefunctions.

- This section also declares all the*user-defined functions*.

Example:

```
int total ;
```

Main () Function Section:

- Every C program must have one main functionsection.
- This section contains two parts; declaration part and executablepart

  1. **Declaration part:** The declaration part declares all the*variables*used in the executablepart.
  2. **Executable part:** There is at least one statement in the executablepart.

- These two parts must appear between the opening and closingbraces.

- The*program execution*begins at the opening brace and ends at the closingbrace.

- The closing brace of the main function is the logical end of theprogram.

- All statements in the declaration and executable part end with asemicolon.

Subprogram section:

- If the program is a*multi-function program*then the subprogram section contains all the*user- defined functions*that are called in the main ()function.
- User-defined functions are generally placed immediately after the main () function, although they may appear in anyorder.

Example of Basic Structure of C Program:

```
/*C basic structure example
to find sum of twonumbers*/      //documentationsection

#include<stdio.h>        //Link section

inttotal;           //Global declarationsection
intsum(int,int);        //Function declaration section
voidmain()          //Mainsection
 {
 int a,b;
 printf("\n Enter the two numbers : ");
 scanf("%d%d",&a,&b);      /* taking two numbers asinput*/
 total=sum(a,b);       /* calling function.The value returned by the function is storedin
total*/
 printf("\n The sum of two numbers is : %d ",total);
 getch();
 }

 int sum ( intnum1,intnum2)      //User defined section
 {
 intresult;        /* defining variable, its scope lies within function */
 result = num1 + num2 ; /*adding twonumbers*/
 return(result);       //definitionsection
 }
```

Output:

```
Enter the two numbers : 1 2

The sum of two numbers is : 3
```

CHARACTER SET:

- Characters are used in forming either words or numbers or even expressions in Cprogramming.

- Characters in C are classified into 4groups:

i. Letters:

In C programming, we can use both uppercase and lowercase letters of English language

Uppercase Letters: A to Z

Lowercase Letters: a to z

8

ii. Digits:

We can use decimal digits from **0 to 9**.

iii. Special Characters:

C Programming allows programmer to use following special characters:

Character	Name	Character	Name
,	Comma	&	Ampersand
.	Period	^	Caret
;	Semicolon	*	Asterisk
:	Colon	-	Minus
?	Question mark	+	Plus
'	Single quote	=	Equal
"	Double quote	<	Less than
!	Exclamation	>	Greater than
|	Vertical bar	(	Left parenthesis
/	Slash	)	Right parenthesis
\	Back slash	[	Left square bracket
~	Tilde	]	Right square bracket
_	Underscore	{	Left curly bracket
$	Dollar	}	Right curly bracket
%	Percentage	#	Hash-Number sign

iv. WhiteSpaces:

In C Programming, white spaces contains:

- BlankSpaces
- Tab
- CarriageReturn
- NewLine

Letters	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z a b c d e f g h i j k l m n o p q r s t u v w x y z
Digits	0 1 2 3 4 5 6 7 8 9
Special Characters	` ~ ! @ # % ^ & * ( ) _ - + = | { } [ ] : ; ' " / ? < > . ,
White Spaces	Blank space, tab, carriage return, new line

CTOKENS:

- C tokens are the basic buildings blocks in C language which are constructed together to write a C program.
- Each and every smallest individual units in a C program are known as Ctokens.

C tokens are of six types. They are,

1. Keywords (eg: int, while),
2. Identifiers (eg: main, total),
3. Constants (eg: 10, 20),
4. Strings (eg: "total", "hello"),
5. Special symbols (eg: (), {}),
6. Operators (eg: +, /,-,*)

KEYWORDS:

- Keywords are reservedwords.
- These meaning cannot bechanged.
- keywords cannot be used as variable names because that would try to change the existing meaning of the keyword, which is notallowed.

- There are total 32keywords.

Auto	double	int	Struct
Break	else	long	Switch
Case	enum	register	Typedef
Const	extern	return	Union
Char	float	short	Unsigned
Continue	for	signed	Volatile
Default	goto	sizeof	Void
Do	if	static	while

IDENTIFIERS:

- Each program elements in a C program are given a name calledidentifiers.
- Names given to identify Variables, functions and arrays are examples for identifiers. eg. x is a name given to integer variable in aboveprogram.

Rules for an Identifier

1. It must begin with an alphabet or an underscore and notdigits.

2. It must contain only alphabets, digits orunderscore.

3. A keyword cannot be used as anidentifier

4. Must not contain whitespace.

5. Only first 31 characters aresignificant.

CONSTANTS:

- Constants refer to fixed values. They are also called as literals.
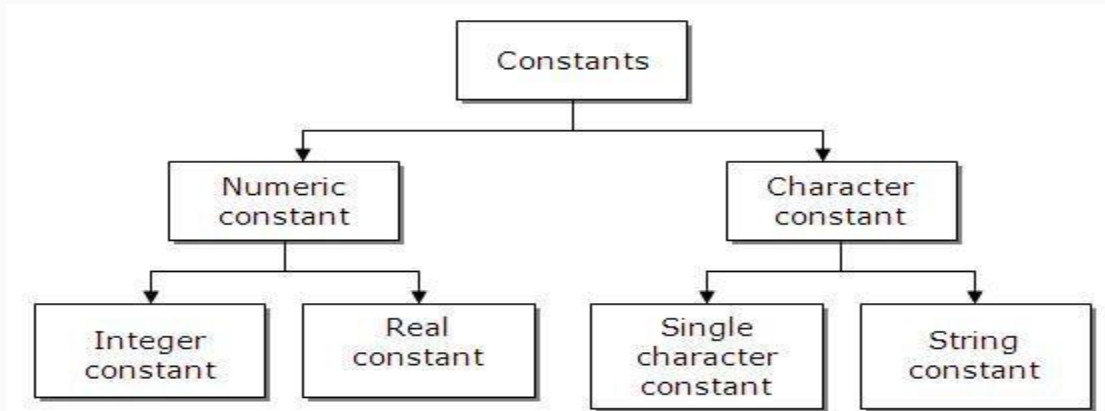- Constants are categorized into two basic types

Fig : Basic types of C constants

Interger Constant:

> ➢ Integer constants are whole numbers without any fractionalpart.
> ➢ Thus integer constant consist of a sequence ofdigits.
> ➢ Integer constant can be written in three different number systems:
>> a) Decimal,
>> b) Octaland
>> c) Hexadecimal.

Decimal Integer:

> ✓ It consists of any combination of digits taken from the set 0 through9.
> ✓ It can either be positive ornegative.
> ✓ ForExample:

0	-321	1234	+786

> ✓ Embedded spaces, commas, and non-digit characters are not permitted betweendigits.
>
> ✓ For example, 15 750 20,000 $10000 are illegalnumbers

OctalInteger:

> ✓ It consist of any combination of digits taken from the set 0 through7.
> ✓ However, the first digit must be 0, in order to identify the constant as an octalnumber.
> ✓ ForExample.

0	01	0125	055

Hexadecimal Integer:

> ✓ A hexadecimal integer constant must begin with either 0x or0X.
> ✓ It can then be followed by any combination of digits taken from the set 0 through 9 and A through F (either upper-case orlower-case).
> ✓ The following are valid hexadecimal integer constants.

0X0	0x1	0XAB125	-0x5bcd

Real Constant:

> ➢ A real constant is combination of a **whole number** followed by a **decimal point** and the **fractional part**. **Example**: 0.0083 -0.75 **.**95215**.**
>
> ➢ Integer numbers are inadequate to represent quantities that vary continuously, suchas distances, heights, temperatures, prices and soon.

11

➢ These quantities are represented by numbers containing fractionalpart.

➢ Such numbers are called **real** or **floating point**constants.

ExponentialNotation

➢ A real number may also be expressed in **exponentialnotation.**

➢ For example, the value 215.65 may be written as 2.1565e2 in exponentialnotation.

➢ e2 means multiply by10^2.

Syntax:

Mantissa e exponent

• Themantissaiseitherarealnumberexpressedindecimalnotationoraninteger.

• The exponent is an integer with an optional plus or minus sign followed by a series ofdigits.

• The letter e separating the mantissa and the exponent can be written in either lowercase or uppercase.

Example: 0.000342 can be represented in exponential form as 3.42e-4

7500000000 can be represented in exponential form as 7.5e9 or75E8

Single Character Constants

➢ It simply contains a single character enclosed within ' and ' (a pair of singlequote).

➢ It is to be noted that the character '8' is not the same as8.

➢ Character constants have a specific set of integer values known as ASCII values (American Standard Code for InformationInterchange).

Example:

'X'   '5'   ';'

String Character Constants

➢ These are a sequence of characters enclosed in double quotes, and they may include letters, digits, special characters, and blankspaces.

➢ Itisagaintobenotedthat"G"and'G'aredifferent-because"G"representsastringasitis enclosed within a pair of double quotes whereas 'G' represents a single character.

Example:

"Hello!",   "2015",   "2+1"

Backslash character constant

➢ Although it consists of two characters, it represents **singlecharacter**.

➢ Each and Every combination starts with **back slash**(),.

➢ They are **non-printable**characters.

➢   They are used in outputfunctions.

For Example:

\t is used to give a tab space

\n is used to give a new line

DATA TYPES:

- Datatypes is a data storage format that can contain a specific type or range ofvalues.
- The kind of data that variables may hold in a programming language are called as datatypes.
- C data types can be classifies into 3 categories.
    1. Primary data types (or fundamental datatypes)
    2. Derived datatypes
    3. User defined datatypes
- All C compiler supports five fundamental data types, namely integer(**int**), character (**char**), floating point (**float**), double-precision floating point (**double**),**void.**
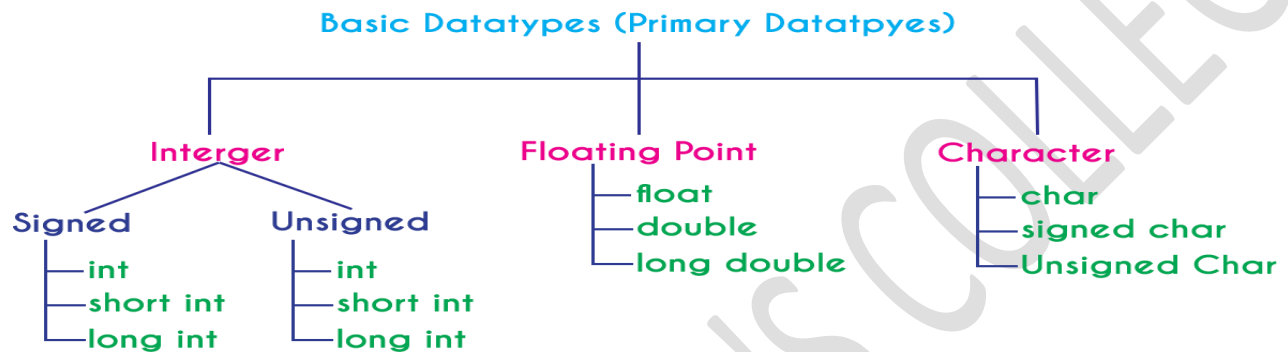
Basic Datatypes (Primary Datatpyes)

Interger

Signed
— int
— short int
— long int

Unsigned
— int
— short int
— long int

Floating Point
— float
— double
— long double

Character
— char
— signed char
— Unsigned Char

Fig: Primary data types in C

Integer Datatype:

➤ Integer datatype is a set of wholenumbers.
➤ Every integer value does not have the decimal value. We use the keyword "**int**" to represent integer data type inc.
➤ We use the keyword **int** to declare the variables and to specify return type of afunction.
➤ The integer data type is used with different type modifiers like short, long, signed andunsigned.
➤ The following table provides complete details about integer datatype.

Type	Size (Bytes)	Range	Specifier
int (signed short int)	2	-32768 to +32767	%d
short int (signed short int)	2	-32768 to +32767	%d
long int (signed long int)	4	-2,147,483,648 to +2,147,483,647	%d
unsigned int (unsigned short int)	2	0 to 65535	%u
unsigned long int	4	0 to 4,294,967,295	%u

13

Floating Point Data Types:
- ➢ Floating point data types are set of numbers with decimalvalue.
- ➢ Every floating point value must contain the decimalvalue.

Float : It stored in 32 bits, with 6 digits of precision. Keyword is **float**.
Double : It uses 64 bits giving a precision of 14 digits. Keyword is **double**.
Long double : It uses 80 bits , precision is above 14 digits. Keyword is**long.**

Type	Size (Bytes)	Range	Specifier
float	4	1.2E - 38 to 3.4E + 38	%f
double	8	2.3E-308 to 1.7E+308	%ld
long double	10	3.4E-4932 to 1.1E+4932	%ld

Character Data Types:
- ➢ Character data type is a set of characters enclosed in singlequotations.
- ➢ A single character can be defines as a character type data. Keyword is**char**.

Type	Size (Bytes)	Range	Specifier
char (signed char)	1	-128 to +127	%c
unsigned char	1	0 to 255	%c

Void Types:
- ➢ The void data type means nothing or novalue.
- ➢ Generally, void is used to specify a function which does not return anyvalue.
- ➢ We also use the void data type to specify empty parameters of afunction.

VARIABLES

- • Variable is a data name that may be used to store a datavalue.
- • variables are changeable, we can change value of a variable during execution of aprogram.
- • A programmer can choose a meaningful variablename.
- • A variable must be declared before it can bedefined.

Example : average, height, age, total etc….

Datatype of Variable:

- • A **variable** in C language must be given a type, which defines what type of data the variablewill hold.

    - ➢ **char**: Can hold/store a character init.

    - ➢ **int:** Used to hold aninteger.

    - ➢ **float:** Used to hold a floatvalue.

    - ➢ **double:** Used to hold a doublevalue.

    - ➢ **void**

14

Rules to name a Variable:

1. Variable name must not start with a digit.

2. Variable name can consist of alphabets, digits and special symbols like underscore _.

3. Blank or spaces are not allowed in variablename.

4. Keywords are not allowed as variablename.

5. Upper and lower case names are treated as different, as C is case-sensitive, so it is suggested to keep the variable in lowercase.

Declaration of variables:

- Declaration of variables must be done before they are used in theprogram.
- Declaration does twothings.
    1. It tells the compiler what the variable nameis.
    2. It specifies what type of data the variable willhold.
- Until the variable is defined the compiler doesn't have to worry about allocating memory space to thevariable.

Syntax:

> **Data_type   v1,v2, ……… vn;**

- ✓ v1,v2,…..vn are the names ofvariables.
- ✓ Variables are separated by commas(**,**).
- ✓ A declaration statement must end with a semicolon(**;**).

For example:

> int number, total;
>
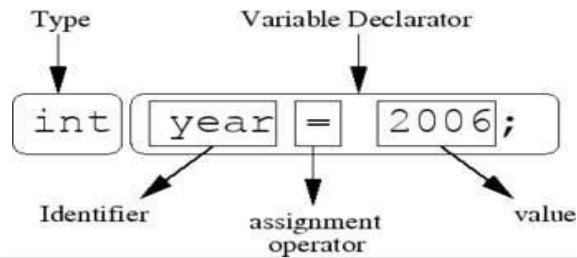> floatprice;
>
> doubleratio;

Initialization of Variable: (Assigning a value to a variable)

- Variable initialization means assigning a value to thevariable.
- C variables declared can be initialized with the help of assignment operator'='.

Syntax

> data_type   variable_name = constant/ expression;
> or
> variable_name = constant /expression;

Example

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

- Multiple variables can be initialized in a single statement by single value.

    For example, a = b = c = d = e =10;

#include<stdio.h>

#include<conio.h>

void main()

{

int a,b, sum;   //variable declaration

a=10;   //variable initialization

b=20;   //variable initialization

sum=a+b;   // executable part

printf("Sum is %d",sum);

getch();

}

Output:

Sum is 30

DECLARATION OF STORAGE CLASS:

- A storage class defines the scope (visibility) and life-time of variables and/or functions within a C Program.
- They precede the type that theymodify.
- We have four different storage classes in a C program−
    1. Automatic Variable (auto).
    2. Register Variable(register).
    3. Static Variable(static).
    4. External Variable(extern).

Syntax: storage_class_specifierdata_typevariable_name;

16

Automatic Variable (auto):
- ➢ Variable defined with **auto**storage class are local to the function block inside which they aredefined.
- ➢ A variable declared inside a function without any storage class specification, is by default an **automaticvariable**.
- ➢ Automatic variables can also be called **local variables** because they are local to afunction.

Syntax: | **auto   data_type   variable_name;**

Example:

```
#include<stdio.h>
void main( )
{
        int detail;
        // or
        auto int detail; // both are same

}
```

Register Variable (register):
- ➢ The variables declared using register storage class specifier are treated similarly like that defined by auto storage class specifier with the only difference is that the variables are stored within the CPU registers providing fasteraccess.
- ➢ It is recommended to use register storage class for variables which are being used at many places. The **register**keyword is used to declare registervariables.

Syntax: | **register   data_type   variable_name;**

Example:

```
#include<stdio.h>
int main()
{
   register int i;
   for(i=1; i<=100; i++)
     printf("n%d",i);
   return 0;

}
```

Static Variable (static):
- ➢ A static variable is declared by using keyword**static.**
- ➢ The variables declared with static storage class specifier are initialized with zero initial value if any initial value is not provided at the time of declaration and it can be accessed from anywhere within the block in which it isdefined.
- ➢ However, the variables declared with static storage class are not destroyed even after program control exits from the block. Thus, the value of the variable persists between different functioncalls.

Syntax: | **static   data_type   variable_name;**

Example:

```
#include<stdio.h>
increment()
{
   static int i=1;
   printf("%dn",i);
   i++;
}
```

17

```
main()
{
   increment();
   increment();
   increment();
   return 0;
}
```
Output
1
2
3

External Variable (extern):

➢ The variable declared using extern storage class are stored in memory with by default zero initial value and continue to stay within the memory until the program's execution is not terminated.

➢ Moreover, variables declared as extern can be accessed by all functions in the program, thus avoiding unnecessary passing of these variables as arguments during functioncall.

➢ It should be noted that the variables declared outside any function definition are treated as variables with extern storageclass.

Syntax: `extern data_typevariable_name;`

Example:

```
#include<stdio.h>

int i;
main()
{
   printf("ni=%d",i);
   increment();
   increment();
   decrement();
   decrement();
}
increment()
{
   i++;
   printf("nOn increment, i=%d",i);
}
decrement()
{
   i--;
   printf("nOn decrement, i=%d",i);
}
```
Output
i=0
On increment, i=1
On increment, i=2
On decrement, i=1
On decrement, i=0

18

DEFINING SYMBOLIC CONSTANTS

- A symbolic constant is name that substitute for a sequence of character that cannot bechanged.
- The character may represent a numeric constant, a character constant, or astring.
- When the program is compiled, each occurrence of a symbolic constant is replaced by its corresponding charactersequence.

> **Syntax:** | **#define symbolic_name value of constant** |
>
> **Example:**
>
> C program consists of the following symbolic constant definitions.
>
> #define PI 3.141593
> #define TRUE 1
> #define FALSE 0

- #define PI 3.141593 defines a symbolic constant PI whose value is 3.141593. When the program is preprocessed, all occurrences of the symbolic constant PI are replaced with the replacement text 3.141593.
- Note that the preprocessor statements begin with a **#symbol**, and are not end with a semicolon. By convention, preprocessor constants are written in**UPPERCASE**.

```
#include<stdio.h>
#include<conio.h>
#define PI 3.14
void main()
{
float    a,b,c,d=PI;
clrscr();
a=100;
b=a*10;
c=b-a;
printf("\na=%f\nb=%f\nc=%f\nPI=%f",a,b,c,d);
getch();
}
```

> **Output:**
>
> a = 100.0000
> b=1000.0000
> c= 900.0000
> PI= 3.14

DECLARING VARIABLE AS CONSTANTS

- ➢ A constant is a declared value that cannot be changed at runtime.
- ➢ Declare a constant with the reserved word const followed by the constant name, type, equal sign, andvalue.
- ➢ We cannot include a constant in a record or other complexstructure.
- ➢ We can include the reserved word const as a modifier on functionparameters.

> **Syntax:** | **const   datatype   variablename = value;** |
>
> **Example:**
>
> const copyrStr String = "Copyright 2007 by CompanyB";
> const myArrayBIN[] = [36, 49, 64];
> const myArray02 BIN[][] = [[1,2,3],[5,6,7]];

19

DECLARING VARIABLE AS VOLATILE

- ➤ Volatile is a qualifier that is applied to a variable when it isdeclared.
- ➤ It tells the compiler that the value of the variable may change at any time-without any action being taken by thecode
- ➤ To declare a variable volatile, include the keyword volatile before or after the data type in the variabledefinition.

For instance both of these declarations will declare foo to be a volatile integer:

Syntax

volatiledatatypevariablename;   (or)
datatype volatilevariablename;

where

volatile   -keyword

datatype   - any valid data type (int, float etc)

variablename - any user definednames

Example

volatile int a;

int volatile a;

OPERATORS AND EXPRESSIONS:

- • The symbols which are used to perform logical and mathematical operations in a C program are called Coperators.
- • These C operators join individual constants and variables to formexpressions.
- • Operators, functions, constants and variables are combined together to formexpressions.
- • Consider the expression A + B * 5. where, +, * are operators, A, B are variables, 5 is constant and A + B * 5 is anexpression.

Types Of C Operators:

C language offers many types of operators. They are,

1.   Arithmeticoperators
2.   Assignmentoperators
3.   Relationaloperators
4.   Logicaloperators
5.   Bit wiseoperators
6.   Conditional operators (ternaryoperators)
7.   Increment/decrementoperators
8.   Specialoperators

Arithmetic Operators:

- ➤ C programming language provides all basic arithmetic operators: +, -, *, / and%.

Operators	Meanings
+	Addition or unary plus
-	Subtraction or unary minus
*	Multiplication
/	Division
%	Modulo division

Table 1: Arithmetic Operators in C

- ➢ The modulo division operation produces the remainder of an integerdivision.
- ➢ Examples: a – b   a+b   a* b   a/ b   a % b   - a * b
  Here **a** and **b** are **variables** and are known as**operands**.
  The modulo division operator **%** cannot be used on floating point data.

i) IntegerArithmetic:
- ❖ When both the operands in a single arithmetic expression such as a+b are integers, the expression is called an integer expression, and the operations is called integerarithmetic.
- ❖ **Example**, if a and b are integers, a=14 andb=4,
  a + b=14   a – b=10   a * b=56   a / b=3   a % b = 2 (remainder ofdivision)

ii) RealArithmetic:
- ❖ An arithmetic operation involving only real operands is called realarithmetic.
- ❖ A real operand may assume values either in decimal or exponentialnotation.
- ❖ The modulo division operator % cannot be used with realoperands.
- ❖ **Example:** x= 6.0 / 7.0=0.857143   and   y = 1.35 + 2.0 =3.35

iii) Mixed- modeArithmetic:
- ❖ When one of the operands is real and the other is integer, the expression is called a mixed-mode arithmeticexpression.
- ❖ **Example:** 15 / 10.0=1.5   20.25 + 20 = 40.25   4 – 1.25 = 2.75

Example program of Arithmetic Operator:

/* Program to Perform Arithmetic Operations in C */

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

void main()
{
int a = 12, b = 3;
int addition, subtraction, multiplication, division, modulus;

clrscr( );

addition = a + b; //addition of 3 and 12
subtraction = a - b; //subtract 3 from 12
multiplication = a * b; //Multiplying both
division = a / b; //dividing 12 by 3 (number of times)
modulus = a % b; //calculation the remainder

printf("Addition of two numbers a, b is : %d\n", addition);
printf("Subtraction of two numbers a, b is : %d\n", subtraction);
printf("Multiplication of two numbers a, b is : %d\n", multiplication);
printf("Division of two numbers a, b is : %d\n", division);
printf("Modulus of two numbers a, b is : %d\n", modulus);

getch( );
}
```

Relational Operator:

- ❖ Relational operators are commonly used to check the relationship between twovariables.
- ❖ If the relation is true then it will return value 1 or if the relation is false then it will return value0.
- ❖ C programming offers 6 relationaloperators.

Operators	Meanings
<	Is less than
<=	Is less than or equal to
>	Is greater than
>=	Is greater than or equal to
==	Is equal to
!=	Is not equal to

Table 2: Relational Operators in C

- ❖ Relational expression is an expression which contains the relationaloperator.
- ❖ Relational operators are most commonly used in decision statements like *if, while* ,etc….
- ❖ Some simple relational expressions are:

   1 <5   9 !=8   2>1+3   10 < 20 is true   20 < 10 isfalse

Example program of relational operataor:
/* C Relational Operations on integers */

```
#include<stdio.h>
#include<conio.h>
void main()
{
 int a = 9, b=4;
 clrscr( );

 printf(" a > b: %d \n ", a > b);
 printf("a >= b: %d \n", a >= b);
 printf("a <= b: %d \n", a <= b);
 printf("a < b: %d \n", a < b);
 printf("a == b: %d \n", a == b);
 printf("a != b: %d \n", a != b);

getch( );
}
```

OUTPUT:

```
a >  b: 1
a >= b: 1
a <= b: 0
a <  b: 0
a == b: 0
a != b: 1
```

Logical Operators:

- ❖ Logical operators are used when more than one conditions are to be tested and based on thatresult, decisions have to bemade.
- ❖ C programming offers three logical operators. Theyare:

Operator	Meaning
&&	Logical AND
||	Logical OR
!	Logical NOT

Table 3: Logical Operator in C

❖ **Forexample:**   a <18 ‖ a>60   a > b && a>c   a = b && x==10

❖ An expression which combines two or more relational expressions is known as logicalexpression.

Operator	Meaning	Example	Result
&&	Logical and	(5<2)&&(5>3)	False
‖	Logical or	(5<2)‖(5>3)	True
!	Logical not	!(5<2)	True

Truth table of logical operators:

a	b	a && b	a ‖ b	!a	!b
false	false	false	false	true	true
false	true	false	true	true	false
true	false	false	true	false	true
true	true	true	true	false	false

Assignment Operators:

❖ Assignment operators are used to assign result of an expression to avariable.

❖ ' = ' is the assignment operator inC.

❖ Furthermore, C also allows the use of shorthand assignmentoperators.

❖ Shorthand operators take theform:

> var op = exp;

    where *var* is a variable, *op* is arithmetic operator, exp is an expression.
    In this case, 'op=' is known as shorthand assignment operator.
    The above assignment

> var op = exp;

is the same as the assignment

> var =var opexp;

Consider an example:

x + =y ;   Here, the above statement means thesameas   **x = x + y;**

Shorthand Assignment Operator:

Simple assignment operator	Shorthand operator
a = a+1	a + =1
a = a-1	a - =1
a = a* (m+n)	a * = m+n
a = a / (m+n)	a / = m+n
a = a %b	a %=b

Increment and Decrement Operators:

❖ C programming allows the use of ++ and – operators which are increment and decrement operators respectively.

❖ Both the increment and decrement operators are unaryoperators.

❖ The increment operator ++ adds 1 to the operand and the decrement operator – subtracts 1 from the operand.

❖ The general syntax of these operatorsare:

> **IncrementOperator:** *m++   or   ++m*;
>
> **Decrement Operator:** *m- - or - - m*;

❖ In the example above, *m++* simply means *m=m+1;* and *m--* simply means*m=m-1;*

❖ Increment and decrement operators are mostly used in for and whileloops.

❖*++m* is known as **prefix** operator and *m++* is known as **postfix** operator.

❖ A**prefix**operator firstly adds 1 to the operand and then the result is assigned to the variable on the left.

❖ A**postfix**operator firstly assigns value to the variable on the left and then increases the operand by 1.

❖ Same is in the case of decrement operator.

Forexample,

$$X=10;$$

$$Y=X++;$$

In this case, the value of Y will be 10 and the value of X will be 11.

Conditional Operator

❖ The operator pair "?" and ":" is known as conditionaloperator.
❖ These pair of operators are ternaryoperators.
❖ The general syntax of conditional operatoris:

> expression1 ? expression2 : expression3 ;

The conditional operator works as follows:
- The first expression conditionalExpression is evaluated first. This expression evaluates to 1 ifit's true and evaluates to 0 if it'sfalse.
- If conditionalExpression is true, expression1 isevaluated.
- If conditionalExpression is false, expression2 is evaluated.

This syntax can be understood as a substitute of if else statement.
For example,   a =3;   b = 5;

Consider an if else statement as:

```
if (a > b)
        x = a ;
else
        x = b ;
```

Now, this if else statement can be written by using conditional operator as:

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

Exampleprogram:

`#include<stdio.h>`
`#include<conio.h>`
`void main( )`
`{`
`int a=10, b=20,c ;`
 	
`clrscr( );`
 	
`c = (a>b) ? a :b;`
`printf(" %d",c);`
 	
`getch();`
`}`
 	
Output:
 	
20
	`#include<stdio.h>`
	`#include<conio.h>`
	`void main( )`
	`{`
	`int a=40, b=20;`
	
	`clrscr( );`
	
	`(a>b) ?printf("TRUE") : printf("FALSE") ;`
	
	`getch();`
	`}`
	
	Output:
	
	TRUE

BitwiseOperators:

- ❖ In C programming, bitwise operators are used for testing the bits or shifting them left orright.
- ❖ The bitwise operators available in Care:

Operators	Meaning
&	Bitwise AND
!	Bitwise OR
^	Bitwise exclusive OR
<<	Shift left
>>	Shift right

Table 4: Bitwise Operators in C

Special Operators

❖ C programming supports special operators like comma operator, sizeof operator, pointer operators (& and *) and member selection operators (. and->).
❖ The comma operator and sizeof operator are discussed in this section whereas the pointer and member selection operators are discussed in latersections.

1. CommaOperator

➢ The comma operator can be used to link the related expressionstogether.
➢ A comma linked expression is evaluated from left to right and the value of the right most expression is the value of the combinedexpression.

   Forexample: x=(a = 2, b=4,a+b)

➢ In this example, the expression is evaluated from left to right. So at first, variable a is assigned value 2, then variable b is assigned value 4 and then value 6 is assigned to the variablex.
➢ Comma operators are commonly used in for loops, while loops, while exchanging values,etc.

2 .Sizeof( ) operator

➢ The sizeof operator is usually used with an operand which may be variable, constant or a data type qualifier.
➢ This operator returns the number of bytes the operandoccupies.
➢ Sizeof operator is a compile timeoperator.

➢ Some examples of use of sizeof operatorare:

> x = sizeof (a);
> y =sizeof(float);

➢ The sizeof operator is usually used to determine the length of arrays and structures when their sizes are not known. It is also used in dynamic memoryallocation.

ARITHMETIC EXPRESSIONS:

• Arithmetic expression in C is a combination of variables, constants and operators written in a propersyntax.
• C can easily handle any complex mathematical expressions but these mathematical expressions have to be written in a propersyntax.
• Some examples of mathematical expressions written in proper syntax of Care:

Algebraic expression	C expression
axb-c	a*b-c
(m+n)(x+y)	(m+n)*(x+y)
$\left[\dfrac{ab}{c}\right]$	a*b/c
$3x^2+2x+1$	3*x*x+2*x+1
$\dfrac{a}{b}$	a/b
$S=\dfrac{a+b+c}{2}$	S=(a+b+c)/2

Evaluation of Expressions:

- Expressions are evaluated using an assignment statement of theform:

$$\text{variable} = \text{expression}$$

- In the above syntax, **variable** is any valid C variablename.
- When the statement like the above form is encountered, the expression is evaluated first andthen the value is assigned to the variable on the left handside.
- All variables used in the expression must be declared and assigned values before evaluationis attempted.
- Examples of expressionsare:

$$x = a*b-c$$
$$y = b/c*a$$
$$z = a-b/c+d$$

- Expressions are evaluated based on operator precedence and associativity rules when anexpression contains more than oneoperator.

C Operator Precedence

- At first, the expressions within parenthesis areevaluated.
- If no parenthesis is present, then the arithmetic expression is evaluated from left toright.
- There are two priority levels of operators inC.

> **High priority :** * /%
> **Low priority :** +   -

- The evaluation procedure of an arithmetic expression includes two left to right passes through the entireexpression.
- In the first pass, the high priority operators are applied as they are encountered and in the second pass, low priority operations are applied as they areencountered.

Suppose, we have an arithmetic expression as:

$x = 9 - 12 / 3 + 3 *2 - 1$

This expression is evaluated in two left to right passes as:

First Pass
Step 1: $x = 9-4 + 3 * 2 - 1$

Step 2: $x = 9 - 4 + 6 - 1$

Second Pass

Step 1: $x = 5 + 6 - 1$
Step 2: $x = 11 - 1$
Step 3: $x = 10$
But when parenthesis is used in the same expression, the order of evaluation gets changed.

For example,

x = 9 – 12 / (3 + 3) * (2 – 1)
When parentheses are present then the expression inside the parenthesis are evaluated first from left to right. The expression is now evaluated in three passesas:

First Pass
Step 1: x = 9 – 12 / 6 * (2 – 1)

Step 2: x= 9 – 12 / 6 * 1

Second Pass
Step 1: x= 9 – 2 * 1

Step 2: x = 9 – 2

Third Pass
Step 3: x= 7

There may even arise a case where nested parentheses are present (i.e. parenthesis inside parenthesis). In such case, the expression inside the innermost set of parentheses is evaluated first and then the outer parentheses are evaluated.

For example, we have an expression as:

x = 9 – ((12 / 3) + 3 * 2) – 1

The expression is now evaluated as:

First Pass:
Step 1: x = 9 – (4 + 3 * 2) – 1

Step 2: x= 9 – (4 + 6) – 1

Step 3: x= 9 – 10 -1

Second Pass
Step 1: x= - 1 – 1

Step 2: x = -2

<u>**Types of conversions**</u>

There are two type of conversions inC.
- Implicit typeconversion
- Explicit typeconversion

Implicit type conversion:

➢ C performs automatic conversions of type in order to evaluate theexpression.
➢ This is called implicit typeconversion.
➢ For example, if we have an integer data type value and a double data type value in an expression then C will automatically convert integer type value to double in order to evaluate theexpression.

Rules for implicit type conversion

Following are the rules for the implicit type conversion in C.

First, all **char**and **short**are converted to **int**data type.

Then,
- If any of the operand in the expression is **long double**then others will be converted to **long double**and we will get the result in **longdouble**.
- Else, if any of the operand is **double**then other will be converted into **double**and the result will be in**double**.
- Else, if any of the operand is **float**then other will be converted into float and the result will be in **float.**
- Else, if any of the operand is **unsigned long int**then others will be converted into **unsigned long int**and we will get the result in **unsigned longint.**
- Else, if any of the operand is **long int**and another is in **unsigned int**then,
  - If **unsigned int**can be converted to **long int**then it will be converted into **long int**and the result will be in **longint**.
  - Else, both will be converted into **unsigned long int**and the result will be in **unsigned long int.**
- Else, if any of the operand is **long int**then other will be converted to **long int**and we will get the result in **longint.**
- Else, if any of the operand is **unsigned int**then other will be converted into **unsigned int**and the result will be in **unsignedint.**

Remember the following hierarchy ladder of implicit type conversion.
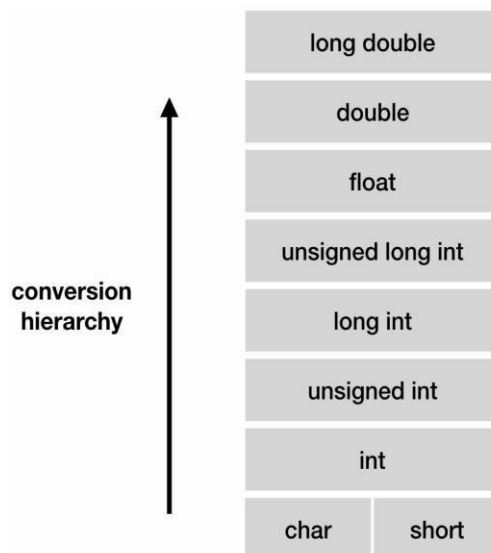
- ➤ If we downgrade from a higher data type to a lower data type then it causes lose ofbits.

- ➤ For example: Moving from double to float causes rounding ofdigits.

- ➤ Downgrading from float to int causes truncation of the fractionalpart.

Explicit type conversion:

- ➤ In explicit type conversion we decide what type we want to convert theexpression.

Syntax

(type) expression

Where, **type** is any of the type we want to convert the **expression** into.

In the following example we are converting floating point numbers into integer.

```c
#include <stdio.h>
int main(void)
{
 //variables
 float
  x = 24.5,
  y = 7.2;

 //converting float to int
 int result = (int) x / (int) y;

 //output
 printf("Result = %d\n", result);

 printf("End of code\n");
 return 0;
}
```

Output

```
Result = 3
End of code
```

In the above code (int) x converts the value 24.5 into 24 and (int) y converts the value 7.2 into 7 so, we get 24/7 i.e., 3 as result because result is of type int and hence the decimal part is truncated.

UNIT–I COMPLETED

MANAGING INPUT AND OUTPUT OPERATIONS

INTRODUCTION:

- Reading, processing, and writing of data are the three essential functions of a computer program.
- The input function **scanf** which can read data from a keyboard.
- The output function **printf** which sends results out to a terminal.
- All input/output operations are carried out through function calls such as **printf** and **scanf**.
- **printf**() and **scanf**() functions are inbuilt library functions in C programming language which are available in C library by default.
- These functions are declared and related macros are defined in **"stdio.h"** which is a header file in C language.
- We have to include **"stdio.h"** file as shown in below C program to make use of these **printf**() and **scanf**() library functions in C language.

READING A CHARACTER:

- Reading a single character can be done by using the function **getchar.**
- The **getchar( )** is used to get or read the input (i.e a single character) at run time.
- The **getchar( )** function accepts any character keyed in.
  Syntax:

  > **variable_name = getchar( );**

  Example:

  > **char name;**
  > **name = getchar( );**

Example Program:

```
#include<stdio.h>
#include<conio.h>
 void main()
{
        char ch;
        ch = getchar();
        printf("Input Char Is :%c",ch);
}
```

WRITING A CHARACTER:

- The **putchar()** function displays the character passed to it on the screen and returns the same character.
- This function too displays only a single character at a time.
- In case you want to display more than one characters, use **putchar()** method in a loop.
  Syntax:

  > **putchar (variable_name);**

  Where **variable_name** is a type char variable containing a character.

  Example:

  > **answer= 'Y';**
  > **putchar (answer);**

        will display the character Y on the screen.
   The another example ,

putchar (' \n ');   where the cursor on the screen move to the beginning of the next line.

Example Program:
```
#include<stdio.h>
#include<conio.h>
void main()
{
        char ch='a';
        putchar(ch);
        getch();
}
```

<div style="border:1px solid">
Output:
a
</div>

FORMATTED INPUT:

- Formatted input refers to an input data that has been arranged in a particular format.
- **For example:**

         15.75   123   rithish

- This line contains three pieces of data arranged in a particular form. The first part of the data should be read into a variable **float,** the second into **int**, and the third part into char.

    Syntax:

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

    ✓ The control string specifies the field format in which the data is to be entered and the arguments arg1, arg2, ……argn specify the address of the locations where the data is stored. Control string and arguments are separated by commas.

a) Inputting Integer Numbers:
   ➢ The field specification for reading an integer number is :**% w sd**
   ➢ The percentage sign (%) indicates that a conversion specification follows.
   ➢ W is an integer number that specifies the field width of the number to be a read.
   ➢ d, known as data type character.

   Example:

scanf(" %2d, %5d", &a, &b);

   Data line: 50   31246
   ✓ The value 50 is assigned to a and 31246 is assigned b.
   ✓ Suppose the input data is as follows: **31246 50**
   ✓ The variable a will be assigned 31 (because of %2d) and b will be assigned 246(unread part of 31246)
   ✓ The value 50 that is unread will be assigned to the first variable in the next **scanf**call.

Note: This kind of error may be eliminated if we use the field specifications without the field which specifications.

   The statement is: scanf ("%d %d", &a, &b);
   will read the data   31246   50
   Correctly and assign 31246 to **a** and 50 to **b**
   ✓ An input field may be skipped by specifying * in the place of field width.
   For example, scanf(" %d %*d %d ", &a, &b)
   Will assign the data **123   456   789**
        as follows:
                        123   to   a
                        456   skipped (because of *)
                        789   to   b

b) Inputting Real Numbers:

  ➢ The field width of real numbers is not to be specified and therefore **scanf** read real numbers using the simple specification **%f**for both the notations, namely, decimal point notations and exponential notation.

  For example:scanf(″ **%f %f %f** ″**, &x, &y, &z);**
  With the input data   475.89   2.156e2   678
  Will assign the value   475.89 to**x**   215.6 to **y**   678.0 to **z.**

If the number to be read is of **double** type, then the specification should be **% lf** instead of simple **% f.**

c) Inputting Character Strings:

  ➢ A **scanf** function can input strings containing more than one character.
  ➢ The specification for reading character strings: **% ws   or % wc**

d) Reading Mixed Data Types:

  ➢ To use one **scanf** statement to input a data line containing mixed mode data.
  ➢ To ensure that the input data items match the control specifications in *order* and *type*.
  ➢ **Example :**
  scanf(″ **%d %c %f %s** ″**, &pid, &code, &price, pname);**

   Will read the data

15	s	54.50 cintol		

e) Detection of Errors in Input:

  ➢ A **scanf** function completes reading its list, it returns the value of number of items that are successfully read.
  ➢ This value can be used to test whether any errors occurred in reading the input.
  For example:

   scanf(″ **%d %f %s** ″**, &a, &b, name);**

Will return the value 3 if the following data is typed in :

   20   150.25motor

   and will return the value 1 if the following line is entered

   20   motor   150.25

  ➢ This is because the function would encounter a string when it was expecting a floating-point value, and would therefore **terminate its scan** after reading the first value.

FORMATTED OUTPUT:

  • The function **printf**() is used for formatted output to standard output based on a format specification.
  • The format specification string, along with the data to be output, are the parameters to the **printf**() function.
  • **Syntax:**

   printf(″**control string** ″ **arg1, arg2, ……. , argn);**

   Control string consists of three types of items:

        1. Characters that will be printed on the screen as they appear.
        2. Format specifications that define the output format for display of each item.
        3. Escape sequence characters such as \n, \t, and \b.

Examples:

printf(″ Programming in c ″);	printf(″ %d ″, a);
printf("*************** ″);	printf(″ X = %d \n Y= %f ″, x, y);
printf("");	printf("Sum= %d ″, x);
printf(″ \n ″);	

a) Outputting Integer Numbers:
  - The format specification for printing an integer number is: $\boxed{\%w\ d}$

Where **w** specifies the minimum field width for the output.

  d specifies that the value to be printed is an integer.

Example:

Format	Output

printf(" %d ", 9876)

9	8	7	6		
		9	8	7	6
9	8	7	6		
		9	8	7	6
0	0	9	8	7	6

printf(" %6d ", 9876)

printf(" %2d ", 9876)

printf(" % -6d ", 9876)

printf(" % 06d ", 9876)

- ✓ The number is written *right- justified* in the given field width.
- ✓ Use minus sign, the number is written in left-justified in the given field width.

b) Output of Real Numbers:
  - The output of a real number may be displayed in decimal notation.

Syntax:

$\boxed{\%\ w.\ p\ f}$

Where **w** – minimum number of positions ,

p – number of digits to be displayed after the decimal point,

Example: y = 98.7654

Format	Output

printf(" % 7.4f ", y)

9	8	.	7	6	5	4
		9	8	.	7	7
9	8	.	7	7		
9	8	.	7	6	5	4

printf(" % 7.2f ", y)

printf(" % -7.2f ", y)

printf(" % f ", y)

c) Printing of a Single Character:
  - A single character can be displayed in a desired position using the format: $\boxed{\%w\ d}$
  - The character will be displayed right- justified in the field of w columns, left-justified by placing a minus sign before the integer w.

d) Printing of Strings :
  - The format specification for outputting strings to that of real numbers.

    Syntax: $\boxed{\%w.ps}$

    Example: String = "NEW DELHI 11001"

Specification   Output

%18s

			N	E	W		D	E	L	H	I		1	1	0	0	1

%18.10s

								N	E	W		D	E	L	H	I	

% .5s

N	E	W		D													

% -18.10s

N	E	W		D	E	L	H	I									

e) Mixed Data Output :
  - It is permitted to mix data types in one **printf** statement.

DECISION MAKING AND BRANCHING

INTRODUCTION:

- C language possesses such decision making capabilities by supporting the following statements.
  - **1. if statement.**
  - **2. Switch statement**
  - **3. Conditional operator statement.**
  - **4. goto statement**

- These statements are popularly known as *decision-making* statements.
- Since, these statements control the flow of execution of a program, there 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.

  Syntax:

if (test expression)

- The expression is evaluated and depending on whether the value of the expression is true (non zero) or false (zero) it transfers the control to a particular statement.
- Some example of decision making, using if statement,

  1) if (room is dark)
       Put on lights

  2) if (age above 18)
       Eligible

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

  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 expression)
{

33

```
                    statement-block;
          }
        statement-x;
```

- ✓ The 'statement-block' may be a single statement or a group of statement.
- ✓ If the test expression is true the statement block will be executed.
- ✓ Otherwise the statement -block will be skipped and the execution will jump to the statement –x.
- ✓ If the condition is true both the statement–block and the statement -x are executed in the sequence .

Flow chart :
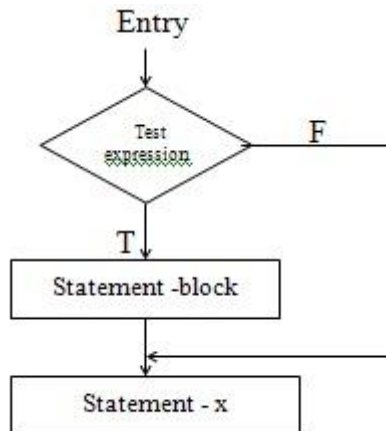
Example:
```
#include <stdio.h>
#include <conio.h>
void main ( )
{
int a = 10;
        if (a <20 )
        {
                printf(" a is less than 20 \n");
        }
        printf( " value of a is : %d " , a);
getch( );
}
```

Output:
a is less than 20
  value of a is : 10

If –Else Statement :
   ✓ The If statement is an extension of the simple If statement.
   Syntax:
```
            if (test expression)
             {
                    true-block statements;
             }
             else
            {
             false-block statements;
            }
             statement – x;
```

34

- If the test expression is true then true-block statement are executed, otherwise the false –block statement are executed.
- In both cases either true-block or false-block will be executed not both.

Flow chart :
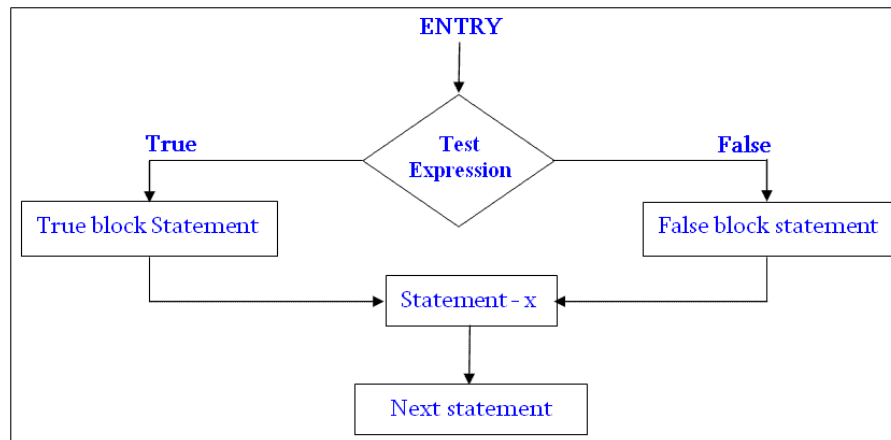
Example:

```
#include <stdio.h>
#include <conio.h>
void main (  )
{
 int x, y ;
     x=15;
     y=18;
         if (x >y )
         {
                 printf(" x is greater than y \n");
         }
         else
         {
                 printf(" y is greater than x \n");
         }
getch( );
}
```
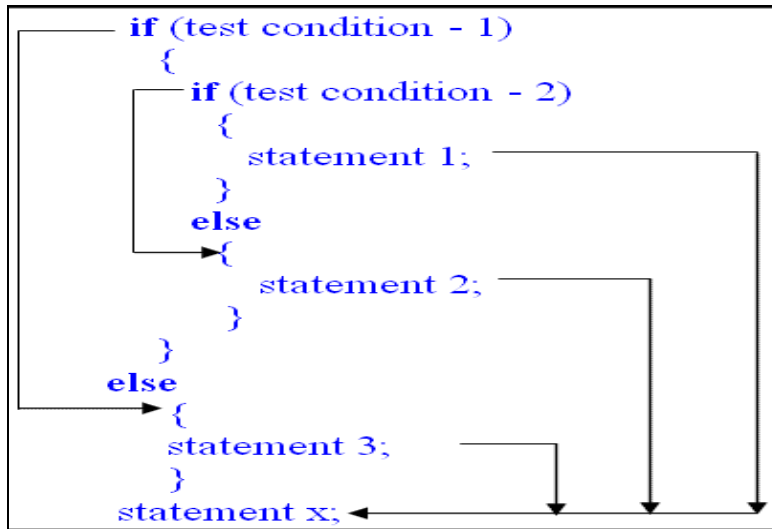
Output:
  y is greater than x

Nested If –else statement :
✓ When a series of decisions are involved we may have to use more than one if-else statement in nested form.
   Syntax:

- ✓ Here, if the condition 1 is false then it skipped to statement 3.
- ✓ But if the condition 1 is true, then it tests condition 2.
- ✓ If condition 2 is true then it executes statement 1 and if false then it executes statement 2.
- ✓ Then the control is transferred to the statement x.

Flow chart:
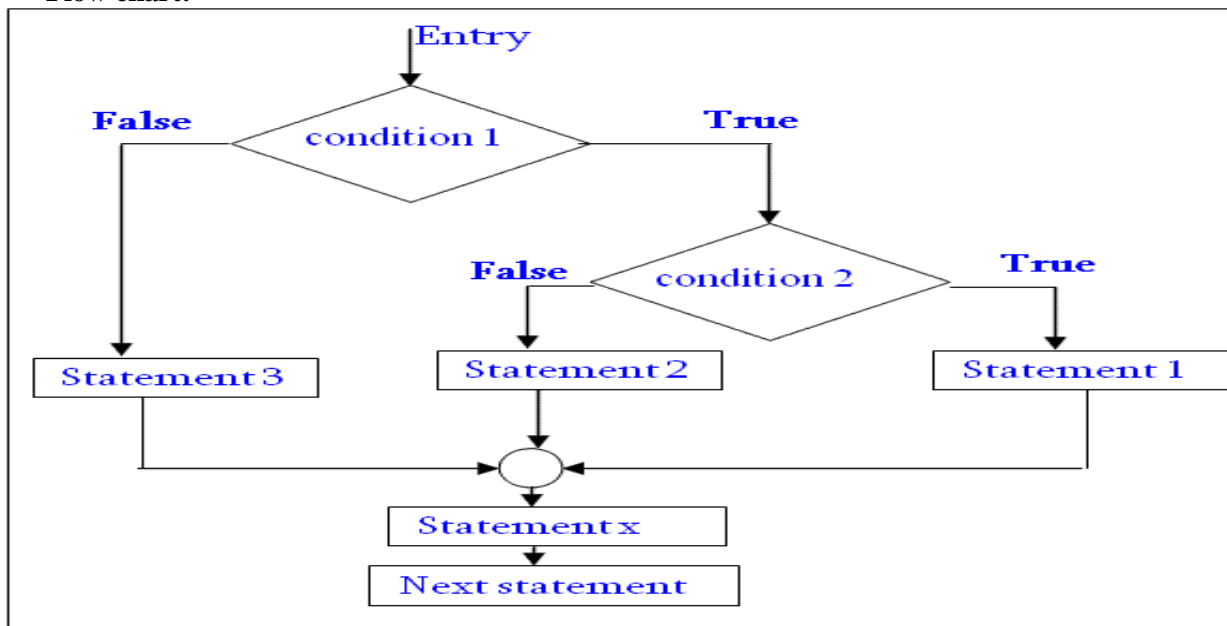
Example:

```
#include <stdio.h>
void main ( )
{
 int a, b, c;
printf( " Enter 3 numbers  :  ");
scanf( " %d %d %d",  &a, &b, &c);
        if ( a> b )
        {
            if (a > c)
                {
```

Output:
Enter 3 numbers : 3   8   4
B is greater

36

```
                              printf( " \n A is greater ");
                              }
                        else
                  {
                              printf(" \n C is greater ");
                              }
                  else
                  {
                        if (b > c)
                              {
                              printf( " \n B is greater ");
                              }
                        else
                  {
                              printf(" \n C is greater ");
                              }
            getch( );
            }
```
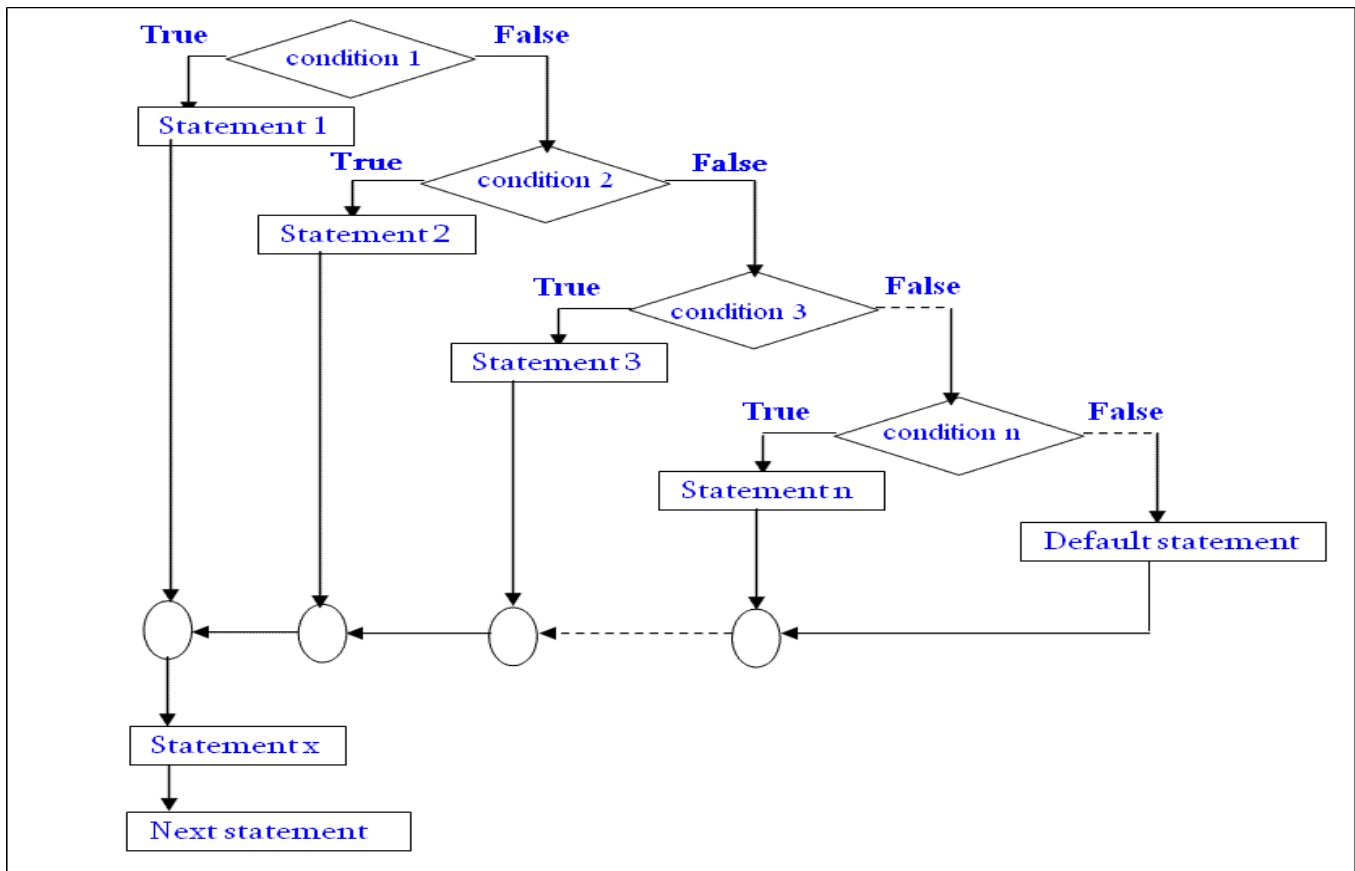
Else-If ladder :
- ✓ When a multipath decision is involved then we use **else if ladder**.
- ✓ A multipath decision is a chain of **if**s in which the statement associated with each **else** is an **if.**
  Syntax:
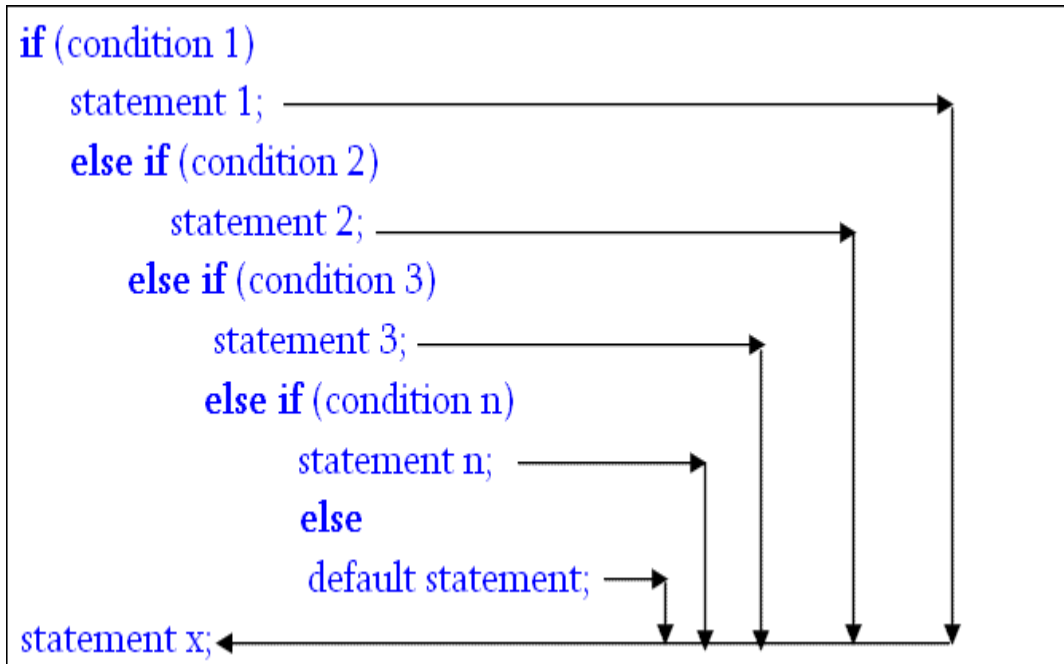
Flow chart:

- ✓ This construct is known as the **else-if** ladder.
- ✓ The conditions are evaluated from the top (of the ladder) down wards. As soon as a true condition is found, the statement associated with it is executed and the control is transferred to the statement-X (i.e.., skipping the rest of the ladder).
- ✓ When all the n-conditions become false, then the final else containing the default-statement will be executed.

Example:

```
#include <stdio.h>
#include <conio.h>
void main ( )
{
 int mark ;
     char   grade;
printf( "\n Enter your mark: ");
scanf( " %d " , &mark);
            if (mark >79 )
                    grade = "Honour";
            else if (mark >59)
                    grade = "First Class";
            else if( mark>49)
                    grade = "Second Class";
            else if( mark>39)
                    grade = "Third Class";
            else
```

Output:
   y is greater than x

```
                    grade = " FAIL";
         printf("\n  %s ", grade);
         getch( );
         }
```
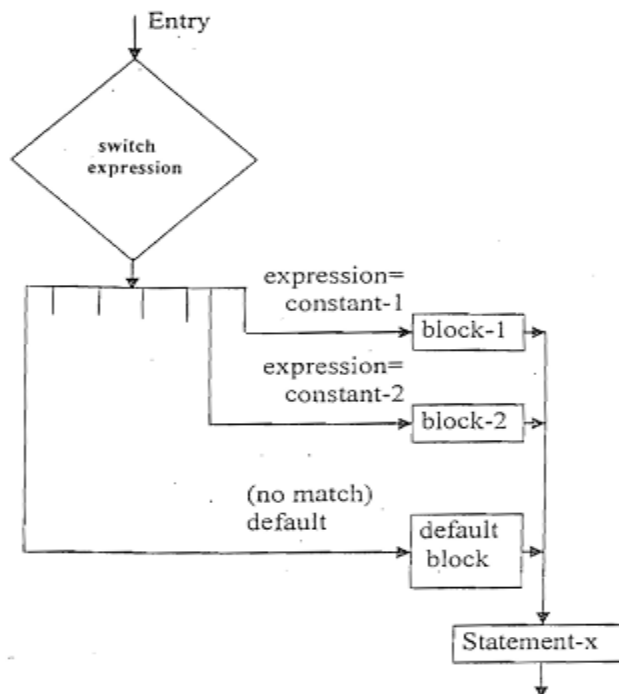
SWITCH STATEMENT :

- The switch statement is a multiway branch statement.
- It provides an easy way to dispatch execution to different parts of code based on the value of the expression.Switch is a control statement that allows a value to change control of execution.

Syntax:

```
         switch (expression)
         {
         case value1   :
                         block1;
                 break;
         case value 2  :
         block 2;
                         break;
         default      :
         default block;
                         break;
           ……….
           ……….
         }
         statement – x;
```

Flow Chart:

39

- ✓ We don't use those expressions to evaluate switch case, which may return floating point values or strings or characters.
- ✓ `break` statements are used to **exit** the switch block. It isn't necessary to use `break` after each block, but if you do not use it, then all the consecutive blocks of code will get executed after the matching block.

`Rules for using switch statement

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

Example :

```
 switch (number)
{
case 1 :printf("Monday");
        break;
case 2 :printf("Tuesday");
        break;
case 3 :printf("Wednesday");
        break;
case 4 :printf("Thursday");
        break;
case 5 :printf("Friday");
        break;
       default :printf("Saturday");
        break;
}
```

Conditional Operator

- ❖ The operator pair "?" and ":" is known as conditional operator.
- ❖ These pair of operators are ternary operators.
- ❖ The general syntax of conditional operator is:

        expression1 ? expression2 : expression3 ;

The conditional operator works as follows:
- The first expression conditionalExpression is evaluated first. This expression evaluates to 1 if it's true and evaluates to 0 if it's false.
- If conditionalExpression is true, expression1 is evaluated.
- If conditionalExpression is false, expression2 is evaluated.

This syntax can be understood as a substitute of if else statement.

For example, $a = 3$; $b = 5$;

Consider an if else statement as:

```
if  (a > b)
        x = a ;
else
        x = b ;
```

Now, this if else statement can be written by using conditional operator as:

        x = (a > b) ? a : b ;

Example program:

| ```c
#include<stdio.h>
#include<conio.h>
void main( )
{
int a=10, b=20,c ;

clrscr( );

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

getch( );
}
``` | ```c
#include<stdio.h>
#include<conio.h>
void main( )
{
int a=40, b=20;

clrscr( );

(a>b) ?printf("TRUE") : printf("FALSE") ;

getch( );
}
``` |
| **Output:**
20 | **Output:**
TRUE |

GOTO STATEMENT :

- The goto statement is used to transfer the control of the program from one point to another.
- It is something referred to as unconditionally branching.
- The goto is used in the form

Goto label;

Label statement :
- ✓ A lable is any valid variable name and must be followed by a colon.
- ✓ We can precode any statement by a label in the form

Label :statement ;

- ✓ This statement immediately transfers execution to the statement labeled with the label identifier.
   A *goto*breaks the normal sequential execution of the program.

Forward Jump: If the *label :* is placed after the*goto* label; some statements skipped and the jump is known as a *forward jump.*

Backward Jump: If the *label :*   is before the statement *goto* label; a loop will be formed and some statements will be executed repeatedly. Such a jump is known as a*backward jump.*

Example:
```c
/* Goto Statement in C Programming example */
#include <stdio.h>
intmain()
```
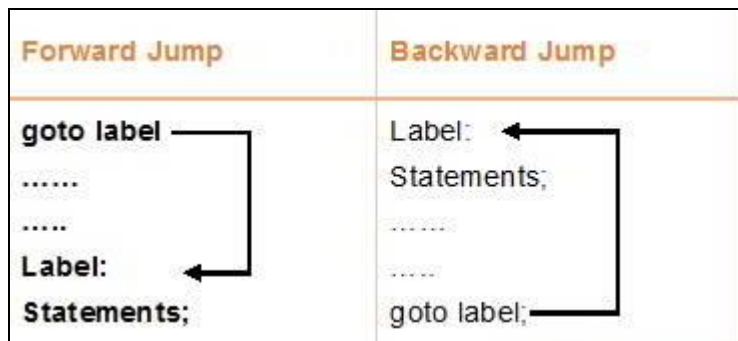
```
{

  intmark;

  printf(" \n Please enter your Mark :  ");
  scanf("%d",&mark);

  if(mark>=50)
   {
    gotoPass;
   }
  else
    gotoFail;

  Pass:
    printf(" \n \n Congratulation! You made it \n");

  Fail:
    printf(" \n \n  Better Luck Next Time \n");

return0;
}
```

DECISION MAKING AND LOOPING

INTRODUCTION:

- Execution of a statement or set of statement repeatedly is called as looping.
- The loop may be executed a specified number of times and this depends on the satisfaction of a test condition.
- A program loop is made up of two parts one part is known as body of the loop and the other is known as control condition.
- Depending on the control condition statement the statements within the loop may be executed repeatedly.
- Depending on the position of the control statement in the loop, a control structure may be classified either as an *entry controlled loop* or as an *exit controlled loop.*

Entry Controlled Loop:

- ✓ When the control statement is placed before the body of the loop then such loops are called as entry controlled loops.
- ✓ If the test condition in the control statement is true then only the body of the loop is executed.

✓ If the test condition in the control statement is not true then the body of the loop will not be executed.
  ✓ If the test condition fails in the first checking itself the body of the loop will never be executed.

Figure: Entry controlled loop   Figure: Exit controlled loop
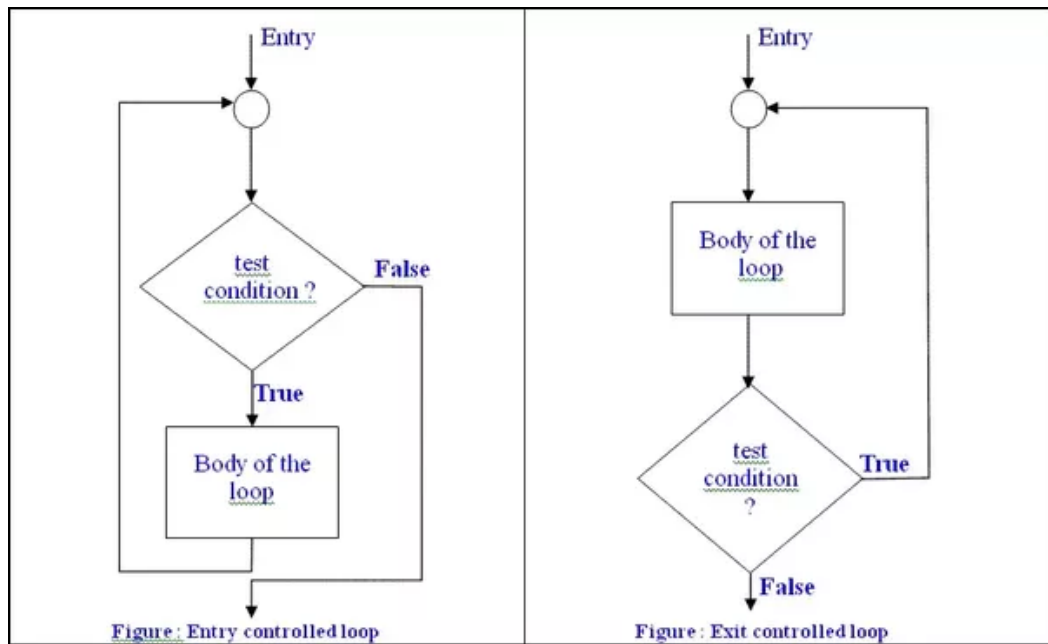
Exit Controlled Loop:

✓ When the control statement is placed after the body of the loop then such loops are called as exit controlled loop.
✓ In this the body of the loop is executed first then the test condition in the control statement is checked.
✓ If it is true then the body of the loop is executed again.
✓ If the test condition is false, the body of the loop will not be executed again. In exit controlled loops even if the test condition fails in the first attempt itself the body of the loop is executed at least once.

Steps of looping process:
• A looping process, in general, would include the following four steps.

1. Setting and initialization of a condition variable.
2. Execution of the statements of the loop.
3. Test for a specified value of the condition variable for execution of the loop.
4. Increasing or updating the condition variable.

Types of Loop:

There are 3 types of Loop in C language, namely:

1. `while` loop

2. `do while` loop

3. `for` loop

WHILE STATEMENT:

- This is an entry controlled loop.
- In this the test condition is placed before the body of the loop.
- If the test expression is true (nonzero), codes inside the body of while loop are executed. The test expression is evaluated again. The process goes on until the test expression is false.
- When the test expression is false, the while loop is terminated.

Syntax:

```
while (test condition)
{
body of the loop;
}
```
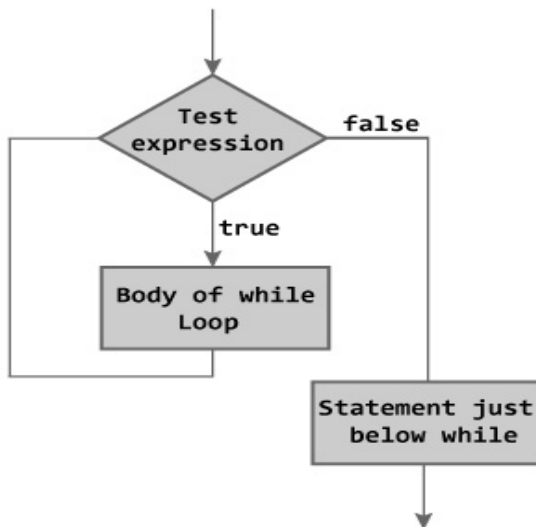
Flow Chart:

Figure: Flowchart of while Loop

Example: Program to print first 10 natural numbers

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

void  main( )
{
int x=1;
clrscr( );

while ( x< = 10 )
{
printf("%d\t", x);

x++;
}

getch( );
}
```

OUTPUT:

1 2 3 4 5 6 7 8 9 10

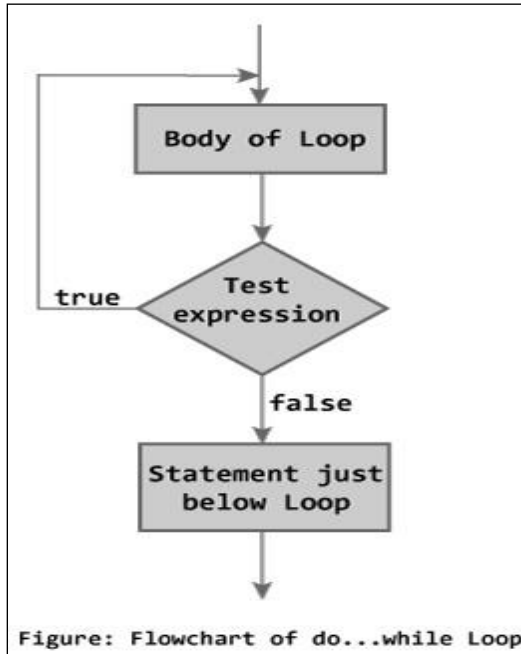## DO - WHILE STATEMENT:

- The do..while loop is similar to the while loop with one important difference.
- The body of do...while loop is executed once, before checking the test expression. Hence, the do...while loop is executed at least once.
- The conditional expression appears at the end of the loop, so the statement(s) in the loop executes once before the condition is tested.
- If the condition is true, the flow of control jumps back up to do, and the statement(s) in the loop executes again. This process repeats until the given condition becomes false.

- **Syntax:**

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

Flow Chart:

Figure: Flowchart of do...while Loop

Example:

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

void  main( )
{
int x=0;
clrscr( );

do
{
printf("%d\t", x);
    x++;
}
while ( x< = 5 )

getch( );
}
```

OUTPUT:

0 1 2 3 4 5

FOR STATEMENT:

A. Simple ' For ' loops:
   o   For loop is an anotherentry controlled loop that provides a more concise loop control structure.
   Syntax:

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

Example:

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

void  main( )
{
int  i;
clrscr( );
```
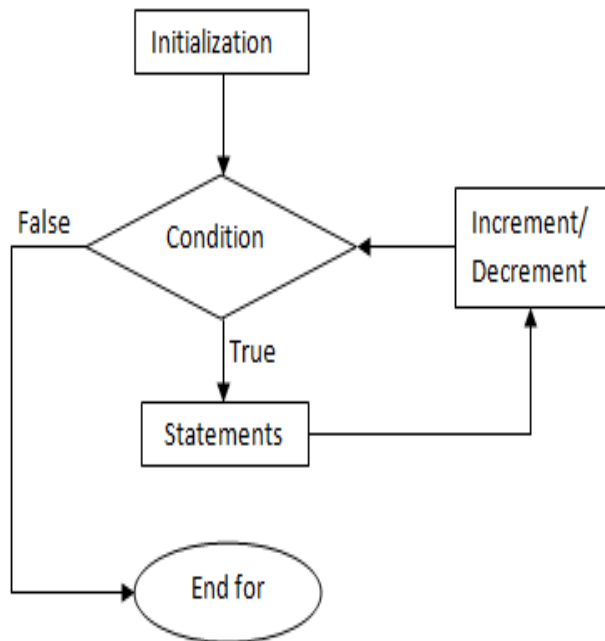
45

Flow Chart:

fig: Flowchart for for loop

```
for ( i= 1; i<=5; i++)
{
printf("%d \t", i);

}

getch( );
}
```

OUTPUT:

1 2 3 4 5

How for loop works?

Step 1: First initialization happens and the initialization statement is executed only once.

Step 2: Then, the test expression is evaluated. If the test expression is false (0), for loop is terminated. But if the test expression is true (nonzero), codes inside the body of `for` loop is executed and the update expression is incremented or decremented, depending on the operation (++ or –).

Step 3: This process repeats until the test expression is false.

Step 4: The `for` loop is commonly used when the number of iterations is known.

B. Additional Features of for loop:

- o   The **for** loop in c has several capabilities that are not found in other loop constructs.
- o   More than one variable can be initialized at a time in the for statement.
- o   **For example:**

```
        p = 1;
        for( n= 0; n<= 1; ++n)
```
Can be rewritten as
```
        for(p=1, n=0; n<=10; ++n);
```
  ✓   It is initializing two variables. **Note:** both are separated by comma (,).

- •   **Another example:**

    The increment section may also have more than one part.
```
    for(n=1 , m= 50 ; n< = m; n = n+1, m = m-1)
    {
        p=m/n;
```

46

printf(%d %d %d \n ", n, m, p);
}
✓ It has two variables in increment part. **Note:** Should be separated by comma.

C. **Nested of For loops:**

- Nested of loops, that is, one for statement within another for statement, is allowed in c.

Syntax:

for(**initialization;** **test – condition ;** **increment** **or** **decrement**)
{
    for(**initialization;** **test – condition ;** **increment** **or** **decrement**)
{
        statement(s);
}

statement(s);
}

Flow Chart:

Example:

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

void  main( )
{
int   i, j ;
clrscr( );

for ( i= 0; i<3; i++)
{
        for ( j= 0; j<3; i++)
{
printf("%d \t  %d \n", i, j);
        }
}
getch( );
}
```

ample 1: Program to print half pyramid using *

```c
#include<stdio.h>
void  main( )
{
int i,j ;
for(i=1; i<=4;++i)
{
for(j=1; j<=i;++j)
{
printf( " * " );
}
printf("\n");
}
}
```

Output:
```
*
*  *
*  *  *
*  *  *  *
```

Example 2: Inverted half pyramid using *

```c
#include<stdio.h>
void  main( )
{
int i,j ;
for(i=4; i>=1;--i)
{
for(j=1; j<=i;++j)
{
printf("* ");
}
printf("\n");
}
}
```

Output:
```
*  *  *  *
*  *  *
*   *
*
```

Example 3: Program to print full pyramid using *

```c
#include<stdio.h>
void  main( )
{
inti,j,n,rows,k;
printf("Input
number of rows : ");
scanf("%d",&rows);
 n=rows+4-1;
for(i=1;i<=rows;i++)
{
for(k=n; k>=1;k--)
{
printf(" ");
}
        for(j=1;j<=i;j++)
        printf("* ");
```

Output:
```
Input number of rows  : 4
    *
   * *
  * * *
 * * * *
```

```
printf("\n");
   n--;
}
}
```
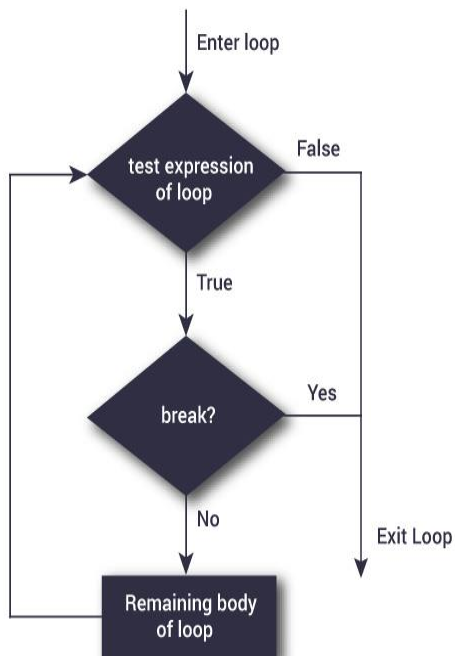
BREAK AND CONTINUE STATEMENT:

Break Statement:
- The break statement is used inside loop or switch statement.
- When **break** statement is encountered inside a loop, the loop is immediately exited and the program continues with the statement immediately following the loop.

Syntax :

> **break;**

Flowchart :

Example:

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

void  main( )
{
int a=0;
     while(a<=5)
      {
if(a= = 3)
        break;

printf("\n Statement %d", a);
        a++;
      }
printf("\n End of Program.");

getch( );
}
```

OUTPUT:
Statement 1
Statement 2
End of Program.

```
while (test expression) {
    statement/s
    if (test expression) {
        break;
    }
    statement/s
}
```

```
do {
    statement/s
    if (test expression) {
        break;
    }
    statement/s
} while (test expression);
```

```
for (intial expression; test expression; update expression) {
    statement/s
    if (test expression) {
        break;
    }
    statements/
}
```

Continue Statement:

- The continue statement is used inside loops, it skips some statements inside the loop.
- When compiler finds the break statement inside a loop, compiler will skip all the following statements in the loop and resume the loop.
- **Syntax :**

  continue;

Flowchart :

Example:

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

void  main( )
{
int a=0;
    while(a<5)
     {
        a++;
if(a= = 3)
        continue;

printf("\n Statement %d", a);
    }
printf("\n End of Program.");
getch( );
}
```

OUTPUT:
Statement 1
Statement 2
Statement 4
Statement 5
End of Program.

```
    ┌──► while (test expression) {         do {
    │         statement/s                      statement/s
    │         if (test expression) {           if (test expression) {
    └───────── continue;              ┌──────── continue;
              }                       │         }
              statement/s             │         statement/s
         }                            │     }
                                      └──► while (test expression);


    ┌──► for (intial expression; test expression; update expression) {
    │         statement/s
    │         if (test expression) {
    └───────── continue;
              }
              statements/
         }
```
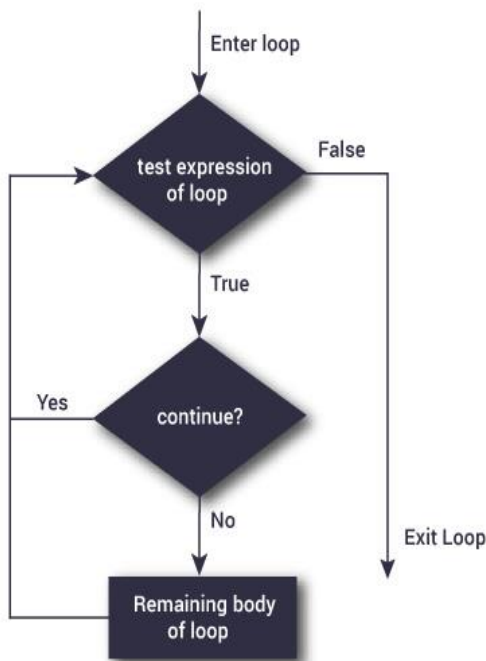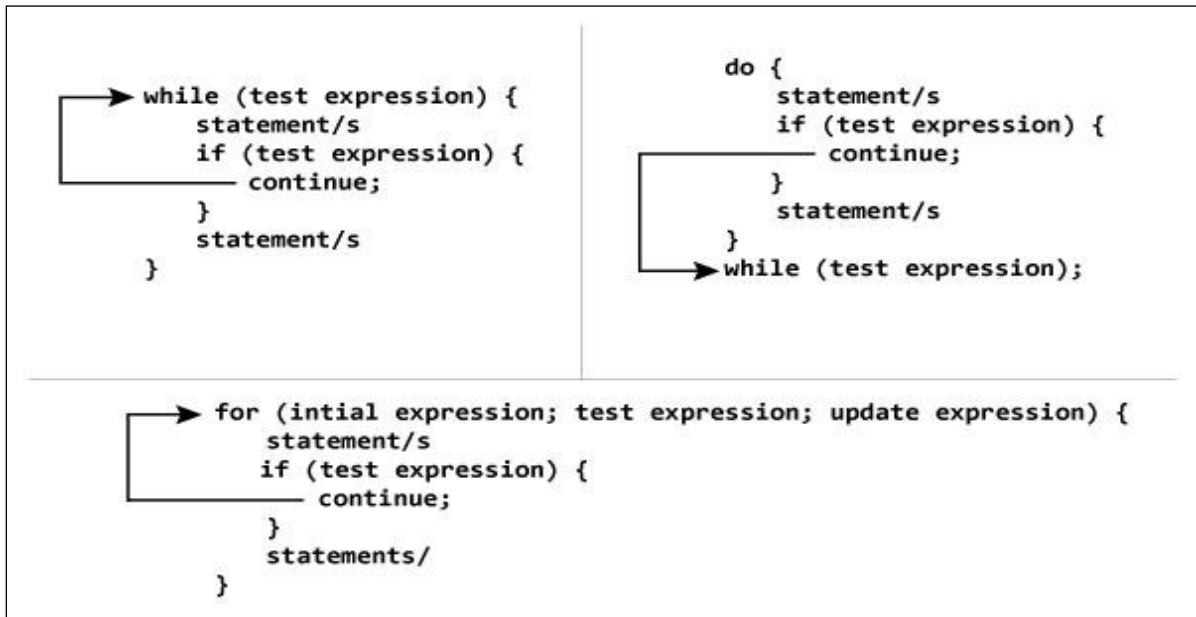
UNIT-II COMPLETED

UNIT-III
ARRAYS

INTRODUCTION:

➢ C Array is a collection of variables belongings to the same data type. You can store group of data of same data type in an array.

- o Array might be belonging to any of the data types

- o Array size must be a constant value.

- o Always, Contiguous (adjacent) memory locations are used to store array elements in memory.

- o It is a best practice to initialize an array to zero or null while declaring, if we don't assign any values to array.

Example where arrays are used,

➢ to store list of Employee or Student names,

➢ to store marks of students,

➢ or to store list of numbers or characters etc.

ADVANTAGES: Array variable can store more than one value at a time where other variable can store   one value at a time.

Example For C Arrays:

int a[10];   // integer array
char b[10];   // character array   i.e. string

Types of C Arrays:

There are 2 types of C arrays. They are,

1. One dimensional array

2. Two dimensional array

3. Multi dimensional array

- Three dimensional array

- Four dimensional array etc…

ONE DIMENSIONAL ARRAY:

✓ A list of items can be given one variable name using only one subscript and such a variable is called a single-subscripted variable or a one-dimensional array.

For Example,

If we want to represent a set of five numbers, say(40,65,89,75,90) by an array variable **mark,** then we may declare the variable **mark** as follows,

int mark[5];

Storage locations:

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

Value to assigned the array elements:

mark[0]= 40;   mark[1] = 65;   mark[2] = 89;   mark[3] = 75;   mark[4] = 90;

Array number to store the value:

mark[0]	mark[1]	mark[2]	mark[3]	mark[4]
40	65	89	75	90

DECLARATION OF ONE - DIMENSIONAL ARRAY:

- Like any other variable, arrays must be declared before they are used so that the compiler can allocate space for them in memory.

Syntax:

type   variable-name[size];

✓ The type specifies the type of element that will be contained in the array, such as **int**, **float**, or **char.**

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

Example:

int salary[10];
float height[5];
char name[15];

✓ Declares the **salary** as an array to contain a maximum of 10 integer constants, any subscripts 0 to 9 are valid.

<div align="center">int a[5]={10,20,30,100,5}</div>

✓ Declares the **height**to be an array containing 50 real elements.

✓ Declares the **name** as a character array(string) variable that can hold a maximum of 15 characters .

Suppose we read the following string constant into the string variable **name**
<div align="center">**"WELL DONE"**</div>
Each character of the string is treated as an element of the array name and is stored in the memory as follows:

'W'	'E'	'L'	'L'	' '	'D'	'O'	'N'	'E'	'\0'

INITIALIZATION OF ONE - DIMENSIONAL ARRAY:

- The different types of initializing arrays:

    1. At Compile time
    a) Initializing all specified memory locations.

    b) Partial array initialization

    c) Initialization without size.

    d) String initialization.

    2. At Run Time

1.Compile Time Initialization:

- We can initialize the elements of arrays in the same way as the ordinary variables when they are declared.

- **Syntax:**

> **type array-name[size]={ list of values};**

a) **Initializing all specified memory locations:-**

✓ Arrays can be initialized at the time of declaration when their initial values are known in advance.

✓ Array elements can be initialized with data items of type int, char etc.

<div align="center">**Ex:- int a[5]={10,15,1,3,20};**</div>

✓ During compilation, 5 contiguous memory locations are reserved by the compiler for the variable a and all these locations are initialized as shown in figure.

a[0]	a[1]	a[2]	a[3]	a[4]
10	15	1	3	20
1000	1002	1004	1005	1008

<div align="center">Fig: Initialization of int Arrays</div>

Example:-

int a[3]={9,2,4,5,6}; //error: no. of initial vales are more than the size of array.

b) **Partial array initialization:-**

- ✓ Partial array initialization is possible in c language.

- ✓ If the number of values to be initialized is less than the size of the array, then the elements will be initialized to zeroautomatically.

Ex:-   int a[5]={10,15};

- ✓ Eventhough compiler allocates 5 memory locations, using this declaration statement; the compiler initializes first two locations with 10 and 15, the next set of memory locations are automatically initialized to 0's by compiler as shown in figure.
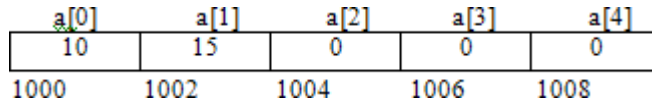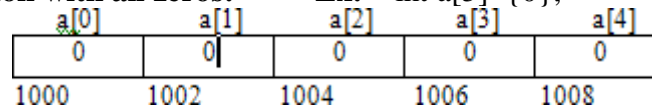
a[0]	a[1]	a[2]	a[3]	a[4]
10	15	0	0	0
1000	1002	1004	1006	1008

Fig: Partial Array Initialization

Initialization with all zeros: Ex: int a[5]={0};

a[0]	a[1]	a[2]	a[3]	a[4]
0	0	0	0	0
1000	1002	1004	1006	1008

c) Initialization without size:-

- ✓ Consider the declaration along with theinitialization.

   Ex:-

char b[]={'C','O','M','P','U','T','E','R'};

- ✓ In this declaration, eventhough we have not specified exact number of elements to be used in array b, the array size will be set of the total number of initial values specified.
- ✓ So, the array size will be set to 8 automatically.
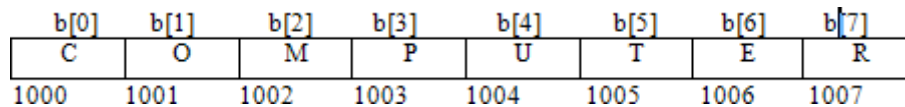- ✓ The array b is initialized as shown in figure.

b[0]	b[1]	b[2]	b[3]	b[4]	b[5]	b[6]	b[7]
C	O	M	P	U	T	E	R
1000	1001	1002	1003	1004	1005	1006	1007

Fig: Initialization without size

   Ex:-   int ch[]={1,0,3,5} // array size is4

d) Array initialization with a string:

- • Consider the declaration with string initialization.

   Ex:-   char b[   ]="COMPUTER";

- • The array b is initialized as shown in figure.

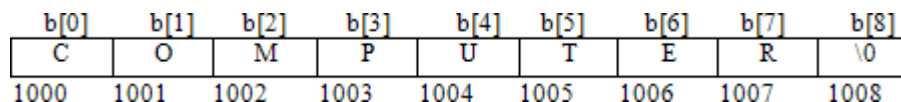| b[0]	b[1]	b[2]	b[3]	b[4]	b[5]	b[6]	b[7]	b[8]
C	O	M	P	U	T	E	R	\0
1000	1001	1002	1003	1004	1005	1006	1007	1008

Fig: Array Initialized with a String

- • Eventhough the string "COMPUTER" contains 8 characters, because it is a string, it always ends with null character. So, the array size is 9 bytes (i.e., string length 1 byte for null character).

Ex:-

> **charb[9]="COMPUTER";   //**
> **correct charb[8]="COMPUTER";**
>   **//wrong**

2. Run TimeInitialization

- An array can be explicitly initialized at run time.

- This approach is usually applied for initializing large arrays.

   Ex:-   scanf can be used to initialize an array.

        int x[3];

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

  The above statements will initialize array elements with the values entered through the key board.

                 (Or)

  for(i=0; i<10; i++)

  {

  printf("\n %d" , i);

  }

TWO – DIMENSIONAL ARRAYS:

- An array consisting of two subscripts is known as two-dimensional array.

- These are often known as array of the array.

- In two dimensional arrays the array is divided into rows and columns, these are well suited to handle the table of data.

DECLARATION OF TWO-DIMENSIONAL ARRAYS:

Syntax: | **data_typearray_name[row_size][column_size];** |

Example: int a[3][3] ;

- Where first index value shows the number of the rows and second index value shows the no. of the columns in the array.

- These are stored in the memory as given below.

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

INITIALIZING TWO DIMENSIONAL ARRAYS:

- Like the one-dimensional arrays, two dimensional arrays may be initialized by following their declaration with a list of initial values enclosed in braces.

   For example,

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

   ✓ Initializes the elements of the first row is zero and the second row is one.

✓ The initialization is done row by row.

- The above statement can be equivalently written as

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

  by surrounding the elements of the each row by braces.
- When the array is completely initialized with all values, explicitly, we need not specify the size of the first dimension.

  That is **,   int table [ ] [3] = { {0,0,0}, {1,1,1}};**   is permitted.

➢ If the values are missing in an initialize, they are automatically set to zero.

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

1	1	0
2	0	0

MULTI-DIMENSIONAL ARRAYS

- **Multidimensional arrays** are often known as array of the arrays.
- In multidimensional arrays the array is divided into rows and columns, mainly while considering multidimensional arrays we will be discussing mainly about two dimensional arrays and a bit about three dimensional arrays.

> **Syntax: data_typearray_name[size1][size2][size3] – [sizeN];**
> **Example: int d[10][20][30];**

Size of multidimensional arrays:

- Total number of elements that can be stored in a multidimensional array can be calculated by multiplying the size of all the dimensions.

  For example:

  The array **int x[10][20]** can store total $(10*20) = 200$ elements.

  Similarly array **int x[5][10][20]** can store total $(5*10*20) = 1000$ elements.

Example program:
/*ADDITION OF TWO MATRICES*/

```
#include <stdio.h>

#include<conio.h>
```

```c
int main()
{
  int m, n, c, d, first[10][10], second[10][10], sum[10][10];
  printf("Enter the number of rows and columns of matrix\n");
scanf("%d%d", &m, &n);
printf("Enter the elements of first matrix\n");
  for (c = 0; c < m; c++)
    for (d = 0; d < n; d++)
scanf("%d", &first[c][d]);
printf("Enter the elements of second matrix\n");
  for (c = 0; c < m; c++)
    for (d = 0 ; d < n; d++)
scanf("%d", &second[c][d]);
printf("Sum of entered matrices:-\n");
  for (c = 0; c < m; c++) {
    for (d = 0 ; d < n; d++) {
      sum[c][d] = first[c][d] + second[c][d];
printf("%d\t", sum[c][d]);
    }
printf("\n");
  }
  return 0;
}
```

```
Enter the number of rows and columns of matrix
2
2
Enter the elements of first matrix
1    2
3    4
Enter the elements of second matrix
5    6
2    1
Sum of entered matrices:-
6    8
5    5
```

CHARACTER ARRAYS AND STRINGS

C STRINGS:

- In C language a string is group of characters (or) array of characters, which is terminated by delimiter \0 (null). Thus, C uses variable-length delimited strings in programs.

- Character strings are often used to build meaningful and readable programs.

- 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.**

Declaring Strings:-
- C does not support string as a data type.

- It allows us to represent strings as character arrays.

- InC,a string variable is any valid C variable name and is always declared as an array of characters.

   Syntax:-

   > **char string_name[size];**

   The size determines the number of characters in the string name
   Example:-

   > **char city[10];**
   > **char name[30];**

Initializing strings: -
- There are several methods to initialize values for string variables.

- For convenience and ease, both initialization and declaration are done in the same step.

Example:

      char c[] = "abcd";   **OR,**

      char c[50] = "abcd";   **OR,**

      char c[] = {'a', 'b', 'c', 'd', '\0'};   **OR,**

      char c[5] = {'a', 'b', 'c', 'd', '\0'};

Reading and Writing strings:-
- ❖ C language provides several stringhandlingfunctionsfor input and output.

String Input/OutputFunctions:-
- ❖ C provides two basic methods to read and write strings. Using formatted input/output functions and using a special set of functions.

Reading strings from terminal:-
(a) Formatted Input Function:-
- ❖ Scanf can be used with %s format specification to read a string.

   Example:
   > **char name[10];**
   > **scanf("%s",name);**
- ❖ Here don"t use „&" because name of string is a pointer to array.
- ❖ The problem with scanfis that it terminates itsinputon the first white space it finds.

Example:

NEW YORK

Reads only NEW (from above example).

(b) Unformatted input functions:

    (1) getchar():-

        o It is used to read a single character from keyboard. Using this function repeatedly we may read entire line of text

    Example:

        char ch="z";

        ch=getchar();

    (2) gets():-

        o It is more convenient method of reading a string of text including blank spaces.

    Example:

        char line[100];

        gets(line);

Writing strings on to the screen:-

    (1) Using formatted output functions:-

        o Printf with %s format specifier we can print strings in different formats on to screen.

    Example:

        char name[10];

        printf("%s",name);

    Example:

        char name[10];

        printf("%0.4",name);

J	A	N	U

/* If name is JANUARY prints only 4 characters ie., JANU */

printf("%10.4s",name);

						J	A	N	U

printf("%-10.4s",name);

J	A	N	U						

    (2) Using unformatted output functions:-

    (a) putchar():-

        o It is used to print a character on the screen.

    Example:

        putchar(ch);

    (b) puts():-

        o It is used to print strings including blank spaces.

    Example:

        char line[15]="Welcome to lab";

        puts(line);

STRING-HANDLING FUNCTIONS

   ❖ C supports a number of string handling functions. Most commonly used string handlings functions are following below.

Function	tion
strcat( )	ncatenates two strings.
strcmp( )	mpares two strings.

strcpy( )	pies one string over another.
strlen( )	ds the length of a string

❖ All of these built- in functions are aimed at performing various operations on strings and they are defined in the header file string.h .

i. strcat( )

❖ This function is used to concatenate two strings. i.e., it appends one string at the end of the specified string.

❖ Its syntax as follows:

> **strcat(string1, string2);**

   where string1 and string2 are one-dimensional character arrays.
❖ This function joins two strings together.

❖ In other words, it adds the string2 to string1 and the string1 contains the final concatenated string. **E.g.,** string1 contains **Prog**and string2 contains **ram**, then string1 holds **program**after execution of the **strcat()** function.

Example:

```
char str1[10 ] = "VERY";
char str2[ 5] ="GOOD";
strcat(string1,string2);
strcat(str1,str2);
```

/* A program to concatenate one string with another using strcat() function*/
```
#include<stdio.h>
#include<string.h>
main()
{
char string1[30],string2[15];
printf("\n Enter first string:");
gets(string1);
printf("\n Enter second string:");
gets(string2);
strcat(string1,string2);
printf("\n Concatenated string=%s",string1);
}
```

Output   1 :
   Enter first string: Good
   Enter second string: Morning
   Concatenated string = GoodMorning

ii. strcmp( )

❖ This function compares two strings character by character (ASCII comparison) and returns one of three values {-1,0,1}.
❖ The numeric difference is **"**0**"**ifstrings are equal .
❖ If it is negative string1 is alphabetically above string2 .
❖ Ifit is positive string2 is alphabetically above string1.

```
char str1[ ] = "WELCOME";
char str2[ ] ="HELLO";
strcpy(str1,str2);
```

Syntax:

strcmp(string1, string2);

Example:

```
char str1[ ] = "ROM";
char str2[ ] ="RAM";
strcmp(str1,str2);
(or)
strcmp("ROM","RAM");
```

/* A program to compare two strings using strcmp()function */
```
#include<stdio.h>
#include<string.h>
void main()
intstrcmp(string1,string2);
{
char string1[30],string2[15];
int x;
printf("\n Enter first string:");
gets(string1);
printf("\n Enter second string:");

gets(string2);
x=strcmp(string1,string2);
if(x==0)
printf("\n Both strings are equal");
else
printf("\n Both strings are  not equal");
}
```

Output   1 :
    Enter first string: Welcome
    Enter second string: Hello
    Both strings are not equal

Output   2:
    Enter first string: Hello
    Enter second string: Hello
    Both strings are equal

iii. strcpy( ) Function:

   ❖ This function is used to copy one string to the other.

   Syntax:

strcpy(string1, string2);

   where string1 and string2 are one-dimensional character arrays.
   ❖ This function copies the content of string2 to string1.

E.g., string1 contains master and string2 contains madam, then string1 holds madam after execution of the strcpy (string1,string2) function.

Example:

/* A program to copy one string to another using strcpy() function */
```
#include<stdio.h>
```

61

```
#include<string.h>
main( )
strcpy(string1,string2);
{
char string1[30],string2[30];
printf("\n Enter first string:");
gets(string1);
printf("\n Enter second string:");
gets(string2);
strcpy(string1,string2);
printf("\n First string=%s",string1);
printf("\n Second string=%s",string2);
}
```

Output:
```
Enter first string: Welcome
Enter second string: Hello
First string     =  Hello
Second string  = Hello
```

  iv. strlen( ) Function:
   ❖ This function is used to find the length of the string excluding the NULL character.
   ❖ In other words, this function is used to count the number of characters in a string.

   Syntax:

   n=strlen(string)

   Example:
```
char str1[ ] = "WELCOME";
int n;
n = strlen(str1);
```

/* A program to calculate length of string by using strlen()function*/
```
#include<stdio.h>
#include<string.h>
void main()
{
char s[20];
int length;
printf("\n Enter any string:");
gets(s);
length=strlen(s);
printf("\n The length of string=%d",length);
}
```

Output:
```
Enter any  string: Welcome
The length of string  = 7
```

USER-DEFINED FUNCTIONS

   ❖ C functions are basic building blocks in a program. All C programs are written using
      functions to improve re-usability, understandability and to keep track on them.

What is C function?
   ❖ A large C program is divided into basic building blocks called C function.

❖ C function contains set of instructions enclosed by "{ }" which performs specific operation in a C program.

❖ Actually, Collection of these functions creates a C program.

Need for User-Defined Functions in C:

➢ Functions are used because of following reasons –
   a) To improve the readability of code.

   b) Improves the reusability of the code, same function can be used in any program rather than writing the same code from scratch.

   c) Debugging of the code would be easier if you use functions, as errors are easy to be traced.

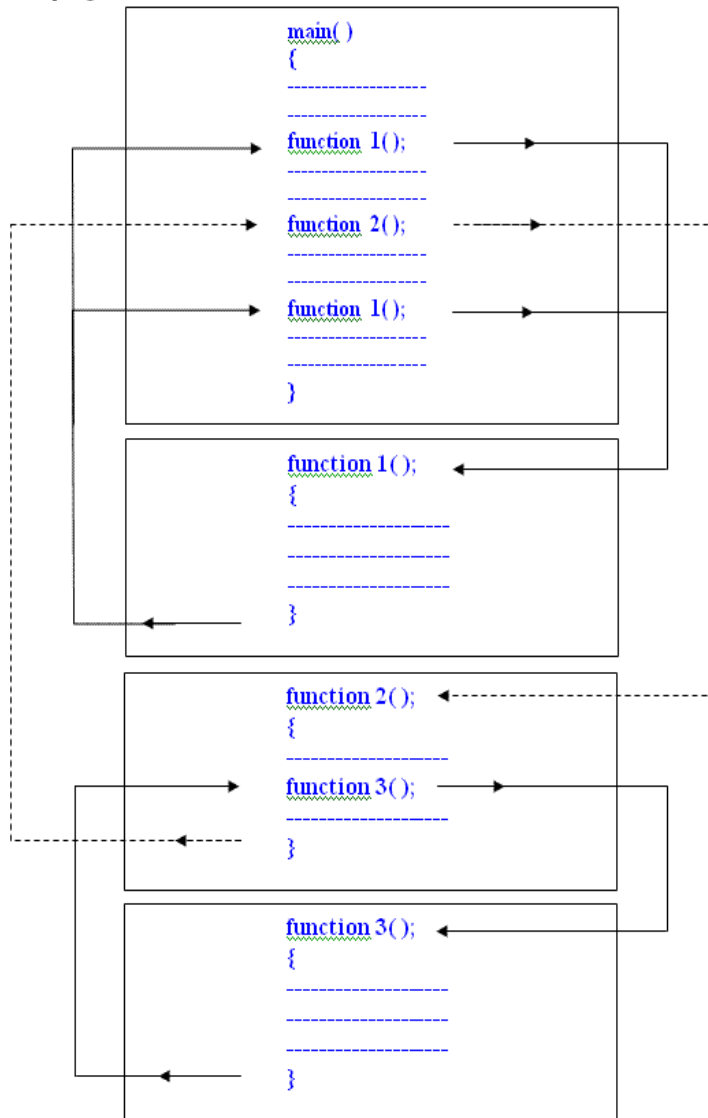   d) Reduces the size of the code, duplicate set of statements are replaced by function calls.

Figure : Flow of control in a multi-function program.

Example Program:

```
#include<stdio.h>
#include<conio.h>
 int add( int x, inty);                    // function prototype

void main( )
{
    int a, b, c;
clrscr( );

printf("Enter two numbers: ");
scanf("%d %d",&a,&b);
```

```
        c = add(a, b);                         // function call
        printf("Addition of two number is  %d", c);
        getch( );
        }

        int add (int x,inty)          // function definition
        {
        int  p;
          p = a+b;
          return p;    // return statement
        }
```

Types of Function in C:
  1) **Library Functions in C:**
    - C provides library functions for performing some operations.
    - These functions are present in the c library and they are predefined.
    - **For example :sqrt( )** is a mathematical library function which is used for finding the square root of any number .The function **scanf( )** and **printf( )** are input and output library function similarly we have **strcmp()** and **strlen()** for string manipulations.
    - To use a library function we have to include some header file using the preprocessor directive **#include**.
    - **For example** to use input and output function like **printf( )** and **scanf( )** we have to include **stdio.h**, for math library function we have to include **math.h** for string library **string.h** should be included.

  2) **User Defined Functions in C**

    - A user can create their own functions for performing any specific task of program are called user defined functions.
    - To create and use these function we have to know these 3 elements.

            1.Function Declaration
            2.Function Definition
            3.Function Call

Function Declaration:
  - Like any variable or an array, a function must also be declared before its used.
  - Function declaration informs the compiler about the function name, parameters is accept, and its return type.
  - The actual body of the function can be defined separately.
  - It's also called as **Function Prototyping**.
  - Function declaration consists of 4 parts.
      - return type (function type).

      - function name.

      - parameter list.

      - terminating semicolon.

```

**Syntax:**

> returntypefunctionName(type1 parameter1, type2 parameter2,...);

**Example:**

> int add (int x, int y);

## Function Definition:

- ➢ The function definition consists of the whole description and code of a function.

- ➢ It tells that what the function is doing and what are the input outputs for that.

- ➢ A function is called by simply writing the name of the function followed by the argument list inside the parenthesis.

## Syntax : Function Definition in C Programming

> return-type function-name(parameters)
>   {
>   declarations
>   statements
>   return value;
>   }

- ➢ Function definitions have two parts:

    1. **Function Header:**

        - ▪ The first line of code is called Function Header.

          **int sum( int x, int y)**
        - ▪ It has three parts:

            i. **Function name**

            ii. **Function type**

            iii. **List of Parameters**

- ❖ **Function name:** It is Unique Name that identifies function. All Variable naming conversions are applicable for declaring valid function name. **E.g: add, sum, etc….,**

- ❖ **Function type:** Return Type is Type of value returned by function. Return Type may be "Void" if function is not going to return a value. **E.g., int, float.**

- ❖ **List of Parameters:** Comma-separated list of types and names of parameters. Parameter injects external values into function on which function is going to operate. Parameter field is optional. If no parameter is passed then no need to write this field. **E.g: int add( int x, int y)**

    2. **Function Body:**

        - ▪ The Function Body contains the declarations and statements necessary for performing the required task.

- The body enclosed in braces { }, contains three parts in the order.

<div align="center">

**i)**      **Local variable declaration**
**ii)**     **Function statement**
**iii)**     **A return statement**

</div>

❖ **Local variable declaration:** That specify the variables needed by the function.

❖ **Function statement:** That perform the task of the function.

❖ **A return statement:** That returns the value evaluated by the function.

## Function Calls:

➢ A function can be called by simply using the function name followed by a list of actual parameters(or arguments).

### Example:

```
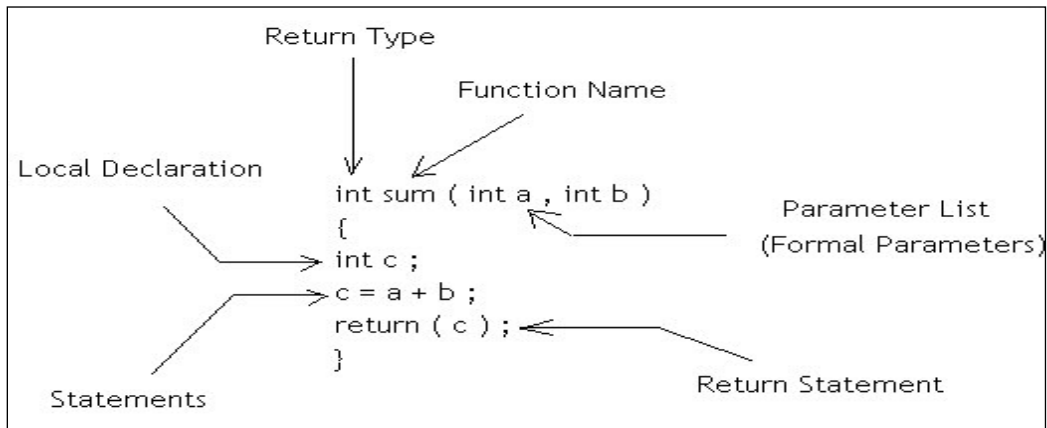main( )
{
    int c;
    c = add(10 , 5);   // function calls
printf("%d\n", c);
 }
```



## CATEGORY OF FUNCTIONS:

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

(i ) Functions with no arguments and no return values.
(ii) Functions with arguments and no return values.
(iii) Functions with arguments and return values.
(iv) Functions with no arguments and return values.
(v) Functions that return multiple values.

**(i) Functions with no arguments and no return values:-**

✓ When a function has no arguments, it does not return any data from calling function.

✓ When a function does not return a value, the calling function does not receive any data from the called function. That is there is no data transfer between the calling function and the called function.

## Example

```
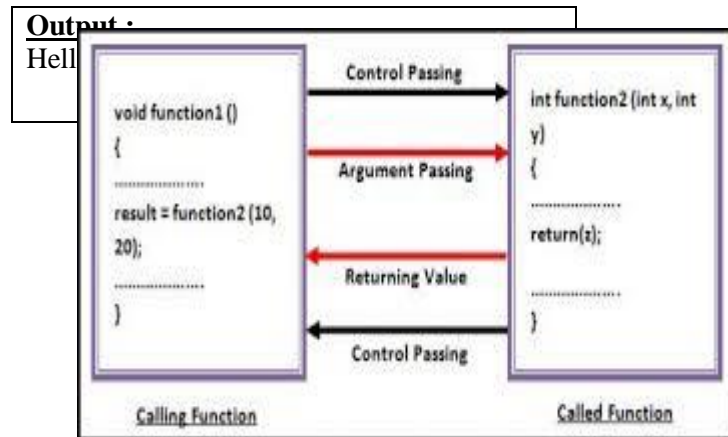#include <stdio.h>
#include <conio.h>
void printmsg()
{
printf ("Hello ! I Am A Function .");
}
  int main()
   {
printmsg( );
return 0;
   }
```

Output :
Hell

Calling Function **Called Function**

(ii) Functions with arguments and no return values:-
  ✓ When a function has arguments data is transferred from calling function to called function.
  ✓   The called function receives data from calling function and does not send back any values
  to calling function. Because it
  doesn't have return value.

Example:

```
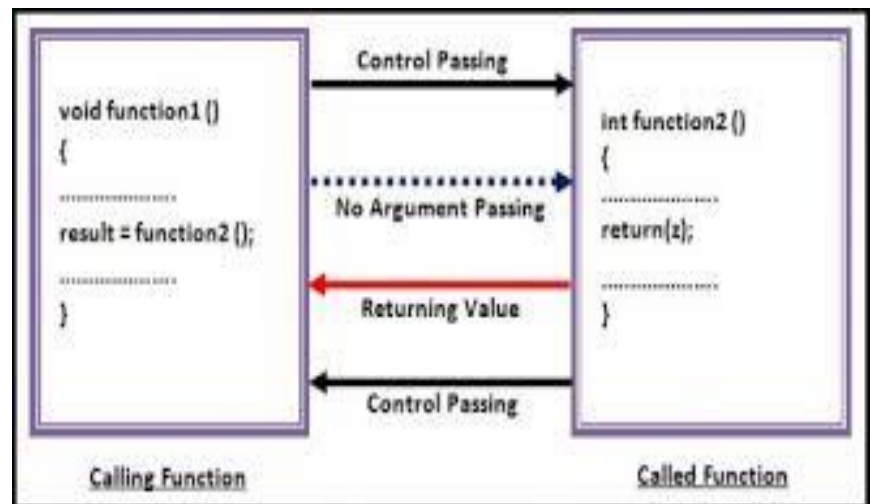#include<stdio.h>
 #include <conio.h>
 void add(int,int);

void main( )

{
int a, b;
printf("Enter two values:
"); scanf("%d%d",&a,&b);
add(a,b);

}
void add (intx, inty)

{
int z ;
 z=x+y;
```


Calling Function **Called Function**

```
            printf ("\n The sum =%d",z);
            }
```

(iii) Functions with arguments and return values:-
- ✓ In this data is transferred between calling and called function.

- ✓ That means called function receives data from calling function and called function also sends the return value to the calling function.

Example:

```
#include<stdio.h>
 #include <conio.h>
 void add(int,int);

 void main()

 {
     int a, b, c;
     printf("Enter two values:
     ");
     scanf("%d%d",&a,&b);
     c=add(a,b);
   printf ("The sum =%d",c);
 }
int add (int x, int y)
{
int z;
z=x+y;
return z;
}
```

(iv) Functions with no arguments but return types:-
- ✓ When function has no arguments data cannot be transferred to called function.

- ✓ But the called function can send some return value to the calling function.

Example:

```
#include<stdio.h>
 #include <conio.h>
 void add(int,int);

 void main( )
 {
     int  c;
     c=add( );
   printf ("The sum =%d",c);
 }
 int add ( )
{
     int x,y,z;
```

```
                    printf("Enter two values: ");
                    scanf("%d%d",&a,&b);
                    z=x+y;
                    return z;
            }
```

NESTING OF FUNCTIONS :

- ➢ It provides a facility to write one function with in another function.

- ➢ There is no limit as to how deeply functions can be nested. This is called nesting of function.

Syntax:
```
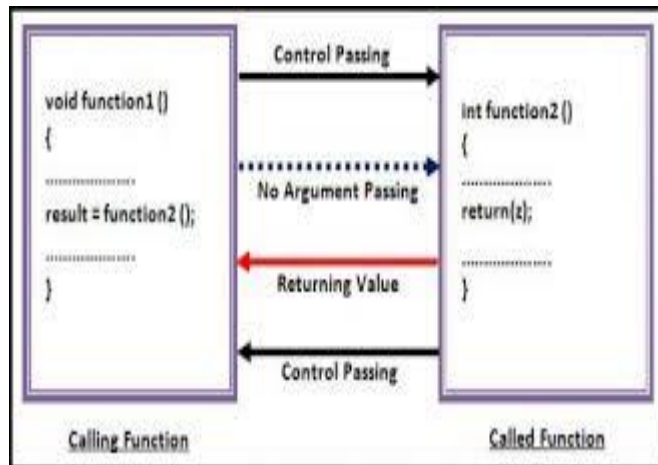main( )
 {
  ………...
  function1( );
  ……….
 }

 function1( );
 {
  ………...
  function2( );
  ……….
 }

 function2( );
 {
  ………...
  ……….
 }
```

➢main () can call Function 1() where Function1 calls Function2() which calls Function3() and so on .

Example:

70

```
main()
{
   clrscr();
   func1();
   getch();
}
void func1()
{
   for(i = 1; i<= 10; i++)
   {
      func2();
   }
}
void func2()
{
   printf("%d\n",i);
}
```

RECURSION:

➢ A function that calls itself is known as a recursive function.

➢ Recursion is the process of repeating items in a self-similar way.

➢ In programming languages, if a program allows you to call a function inside the same function, then it is called a recursive call of the function.

Example:

main( ) **{** **printf("Hello world");** **main( );** **}**	**Output:** **Hello world** **Hello world** **Hello world…..**

- In this program, we are calling *main()* from *main()* which is recursion. But we haven't defined any condition for the program to exit. Hence this code will print "**Hello world**" infinitely in the output screen.

How does recursion work?

> Another useful example of recursion is the evaluation of factorials of a given numbers.

> The factorial of a number n is expressed as a series of repetitive multiplications as shown below.

Factorial of n = n(n-1)(n-2) …….1.
For example: Factorial of 4 = 4*3*2*1 = 24

```
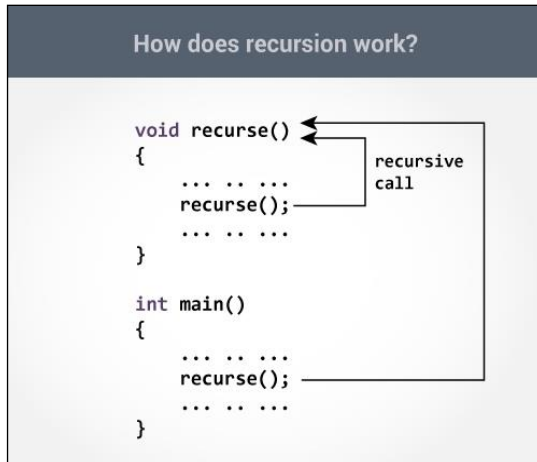#include<stdio.h>
int factorial(int n)
{
   if(n==0)
      return 1;
   else
      return (factorial(n-1)*n);
}
int main()
{
   int num,f;
printf("Enter a number: ");
scanf("%d",&num);
   f=factorial(num);
printf("Factorial of %d = %d",num,f);
   return 0;
}
```

THE SCOPE, VISIBILITY AND LIFETIME OF VARIABLES:

- A storage class defines the scope (visibility) and life-time of variables and/or functions within a C Program. They precede the type that they modify.
- We have four different storage classes in a C program −
     1. Automatic Variable (auto).
     2. Register Variable (register).
     3. Static Variable (static).
     4. External Variable (extern).

Syntax: storage_class_specifierdata_typevariable_name;

Automatic Variable (auto):

➢ Variable defined with **auto** storage class are local to the function block inside which they are defined.

➢ A variable declared inside a function without any storage class specification, is by default an **automatic variable**.

➢ Automatic variables can also be called **local variables** because they are local to a function.

Syntax: auto   data_typevariable_name;
Example:

```
#include<stdio.h>
void main( )
{
        int detail;
        // or
        auto int detail;   // both are same
}
```

Register Variable (register):

➢ The variables declared using register storage class specifier are treated similarly like that defined by auto storage class specifier with the only difference is that the variables are stored within the CPU registers providing faster access.

➢ It is recommended to use register storage class for variables which are being used at many places. The **register** keyword is used to declare register variables.

Syntax: **registerdata_typevariable_name;**
Example:

```
#include<stdio.h>
int main()
{
    registerinti;
    for(i=1; i<=100; i++)
        printf("n%d",i);
    return0;
}
```

Static Variable (static):

➢ A static variable is declared by using keyword **static.**

- The variables declared with static storage class specifier are initialized with zero initial value if any initial value is not provided at the time of declaration and it can be accessed from anywhere within the block in which it is defined.
- However, the variables declared with static storage class are not destroyed even after program control exits from the block. Thus, the value of the variable persists between different function calls.

Syntax: static data_typevariable_name;
Example:

```
#include<stdio.h>
increment()
{
    staticinti=1;
    printf("%dn",i);
    i++;
}
main()
{
    increment();
    increment();
    increment();
    return0;
}
```

Output
1
2
3

External Variable (extern):

- The variable declared using extern storage class are stored in memory with by default zero initial value and continue to stay within the memory until the program's execution is not terminated.
- Moreover, variables declared as extern can be accessed by all functions in the program, thus avoiding unnecessary passing of these variables as arguments during function call.
- It should be noted that the variables declared outside any function definition are treated as variables with extern storage class.

Syntax: extern   data_typevariable_name;

Example:

```
#include<stdio.h>
 inti;
main()
{
    printf("ni=%d",i);
    increment();
    increment();
    decrement();
    decrement();
}
increment()
{
    i++;
    printf("nOn   increment,
i=%d",i);
}
decrement()
{
    i--;
    printf("\nOn
decrement, i=%d",i);
}
```

Output
i=0
On increment, i=1
On increment, i=2
On decrement, i=1
On decrement, i=0

PASSING PARAMETERS TO FUNCTIONS

When a function is called, the calling function has to pass some values to the called functions.

There are two ways by which we can pass the parameters to the functions:

1. Call by value:

- Here the values of the variables are passed by the calling function to the called function.

- If any value of the parameter in the called function has to be modified the change will be reflected only in the called function.

- This happens as all the changes are made on the copy of the variables and not on the actual ones.

Example: Call by value

```
#include <stdio.h>
int sum (int n);
void main( )  {
    int a = 5;
    printf("\n The value of 'a' before the calling function is = %d", a);
    a = sum(a);
    printf("\n The value of 'a' after calling the function is = %d", a);
```

```
}
int sum (int n)   {
    n = n + 20;
    printf("\n Value of 'n' in the called function is = %d", n);
    return n;
}
```

Output:
The value of 'a' before the calling function is = 5
Value of 'n' in the called function is = 25
The value of 'a' after calling the function is = 25

2. Call by reference

- Here, the address of the variables are passed by the calling function to the called function.

- The address which is used inside the function is used to access the actual argument used in the call.

- If there are any changes made in the parameters, they affect the passed argument.

- For passing a value to the reference, the argument pointers are passed to the functions just like any other value.

Example: Call by reference

```
#include <stdio.h>
int sum (int *n);
void main()
{
    int a = 5;
    printf("\n The value of 'a' before the calling function is = %d", a);
    sum(&a);
    printf("\n The value of 'a' after calling the function is = %d", a);
}
int sum (int *n)
{
    *n = *n + 20;
    printf("\n value of 'n' in the called function is = %d", n);
}
```

Output:
The value of 'a' before the calling function is = 5
value of 'n' in the called function is = -1079041764
The value of 'a' after calling the function is = 25

UNIT-III COMPLETED

STRUCTURES AND UNIONS

DEFINITION:
➢ A structure is a collection of one or more variables of different data types, grouped together under a single name. By using structures variables, arrays, pointers etc can be grouped together.
➢ A Structure is a convenient tool for handling a group of logically related data items.
➢ For example, it can be used to represent a set of attributes, such as student_name, roll_no& marks.

DEFINING A STRUCTURE:
➢ The structure definition associated with the structure name is referred as tagged structure.
➢ It doesn't create an instance of a structure and does not allocate any memory.

Syntax of structure:	**Example of structure:**

struct   tag_name
        {
               data_type   member1;
               data_type   member2;
               .
               .
               .
        };
	struct student
	        {
	               char name[50];
	               int age;
	               float height;
	        };

Where,
- struct is the keyword which tells the compiler that a structure is being defined.
- Tag_name is the name of the structure.
- variable1, variable2 … are called members of the structure.
- The members are declared within curly braces.
- The closing brace must end with the semicolon.

DECLARING STRUCTURE VARIABLES:
➢ After defining a structure format we can declare variables of that type.
➢ A structure variable declaration is similar to the declaration of variables of any other data types.
➢ It includes the following elements:
- The keyword **struct**.
- The structure tag name.
- List of variable name separated by **commas**.
- A terminating **semicolon**.

For example,

> **struct   student, stu1, stu2, stu3;**

Declares stu1, stu2 and stu3 as variables of type **struct student.**

Declaring Structure variables separately:

struct   student
        {
               char   name[50];
               int   age;
               float   height;
        };
struct   student, stu1, stu2, stu3;   // declaring variable of struct student

Declaring Structure variables with structure definition:

```
struct    student
        {
                char    name[50];
                int     age;
                float   height;
        } stu1,  stu2,  stu3;     // declaring variable of struct student
```

Here `stu1` and `stu2` are variables of structure `Student`

ACCESSING STRUCTURE MEMBERS

- ➢ Structure members can be accessed and assigned values in a number of ways.
- ➢ Structure members have no meaning individually without the structure.
- ➢ In order to assign a value to any structure member, the member name must be linked with the **structure** variable using a dot [**.**] operator also called **period** or **member access** operator.

For Example,

stu1.name	stu2.name	stu3.name
stu1.	stu2.age	stu3.age
stu1.height	stu2.height	stu3.height

Assign the value to the member of stu1:

```
strcpy(stu1.name, "Varsha");

stu1.age = 20;

stu1.height = 5.5;
```

We can also use `scanf()` to give values to structure members through keyboard.

```
scanf( " %s " , stu1.name);

scanf( " %s " , &stu1.age);
```

STRUCTURE INITIALIZATION:

- ➢ C language does not permit the initialization of individual structure members within the template.
- ➢ The initialization must be done only in the declaration of the actual variables..

Syntax:

> **struct tag_name variable = {value1, value2,… value-n};**

Structure initialization can be done in any one of the following ways.

Example:

```
struct    student
{
        char    name[10];
        int     age;
        float   height;
```

```
} ;
struct  studentstu = { "Dhars",  22,  6.0 } ;
```

Rules for compile time initialization of a variable:

1. The keyword **struct.**
2. The structure tag name.
3. The name of the variable to be declared .
4. The assignment operator = .
5. A set of values for the members of the structure variable, separated by commas and enclosed in braces.
6. A terminating semicolon.

Initializing Multiple Variables:

```
struct    student
{
            char     name[10];
            int      age;
            float    height;
};
struct  student  stu1 = { "Dhars",  22,  6.0 } ;
struct  student  stu2 = { "Sujay",  20,  5.3 } ;
struct  student   stu3 = { "Ajay",  21,  5.5 } ;
```

Initialization along with structure definition:

- Consider the structure definition for student with three fields name, age and height.
- The initialization of variable can be done as shown below,

```
struct    student
{
            char     name[10];
            int      age;
            float    height;
}  stu = { "Dhars",  22,  6.0 } ;
```

ARRAYS OF STRUCTURES:

➢ An <u>array</u> is a collection of elements of same data type that are stored in contiguous memory locations.
➢ A <u>structure</u> is a collection of members of different data types stored in contiguous memory locations.
➢ An <u>array of structures</u> is an array in which each element is a structure.
➢ This concept is very helpful in representing multiple records of a file, where each record is a collection of dissimilar data items.
➢ As we have an array of integers, we can have an array of structures also.

Syntax:

```
struct   tag_namearrayofstructure[size];
```

Let's take an example, to store the information of 3 students, we can have the following structure definition and declaration,

```
struct    student
{
            char     name[10];
            int      age;
            float    height;
};
```

> **struct student stu[3];**

- Defines an array called stu, which contains three elements. Each element is defined to be of type struct student.
- For the student details, array of structures can be initialized as follows,

> **struct studentstu[3]={{"Dhars", 18 ,5.7},{"kani", 21, 5.8},{"Ramya",18, 5.5}};**

ARRAYS WITHIN STRUCTURE
- It is also possible to declare an array as a member of structure, like declaring ordinary variables.
- For example to store marks of a student in three subjects then we can have the following definition of a structure.

```
struct    student
{
          char    name[10];
          int     rollno;
          int     marks[3];
};
struct student stu;
```

- Then the initialization of the array marks done as follows,

> **struct student stu= {"Deepika", 4 , {60,70,80}};**

- The values of the member marks array are referred as follows,
  stu.marks [0] --> will refer the 0th element in the marks
  stu.marks [1] --> will refer the 1st element in the marks
  stu.marks [2] --> will refer the 2ndt element in the marks

NESTED STRUCTURE (STRUCTURE WITHIN STRUCTURE) :
- A structure which includes another structure is called nested structure or structure within structure. i.e a structure can be used as a member of another structure.
- There are two methods for declaration of nested structures.
  - **(i)   The syntax for the nesting of the structure is as follows**

```
                struct tag_name1
                {
                      type1 member1;
                      …….
                      …….
                };
                struct tag_name2
                {
                       type1 member1;
                      ……
                      ……
                      struct tag_name1 var;
                      ……
                };
```

The syntax for accessing members of a nested structure as follows,

(ii)   **The syntax of another method for the nesting of the structure is as follows**

```
struct structure_nm
{
        <data-type>    element 1;
        <data-type>    element 2;
        - - - - - - - - - - - - - - - - - - - - -
        <data-type>    element n;

        struct structure_nm
        {
                <data-type>    element 1;
                <data-type>    element 2;
                - - - - - - - - - - - - - - - - - - - - -
                <data-type>    element n;
        }inner_struct_var;

}outer_struct_var;
```

Example :

```
struct stud_Res
{
        int rno;
        char nm[50];
        char std[10];
        struct stud_subj
        {
                char subjnm[30];
                int marks;
        }subj;
}result;
```

In above example, the structure stud_Res consists of stud_subj which itself is a structure with two members. Structure stud_Res is called as 'outer structure' while stud_subj is called as 'inner structure.'
The members which are inside the inner structure can be accessed as follow :
                    result.subj.subjnm
            result.subj.marks

STRUCTURES AND FUNCTIONS:

➢ Of course a sensible alternative to writing out the addition each time is to define a function to do the same job - but this raises the question of passing structures as parameters.
➢ Fortunately this isn't a big problem.

- Most C compilers, will allow you to pass entire structures as parameters and return entire structures.
- As with all C parameters structures are passed by value and so if you want to allow a function to alter a parameter you have to remember to pass a *pointer* to a **struct**.

Given that you can pass and return **structs** the function is fairly easy:

```
struct comp add(struct comp a , struct comp
b)
{
struct comp c;
c.real=a.real+b.real;
c.imag=a.imag+ b.imag;
return c;
}
```

After you have defined the add function you can write a complex addition as:

```
x=add(y,z)
```

which isn't too far from the **x=y+z** that you would really like to use. Finally notice that passing a **struct** by value might use up rather a lot of memory as a complete copy of the structure is made for the function.

UNIONS:
- A union is one of the derived data types. Union is a collection of variables referred under a single name.
- The syntax, declaration and use of union is similar to the structure but its functionality is different.

Syntax of unions:	Example of structure:
union union_name	union techno
{	{
data_type member1;	int comp_id;
data_type member2;	char nm;
.	float sal;
.	}tch;
.	
}union _variable;	

- A union variable can be declared in the same way as structure variable.

```
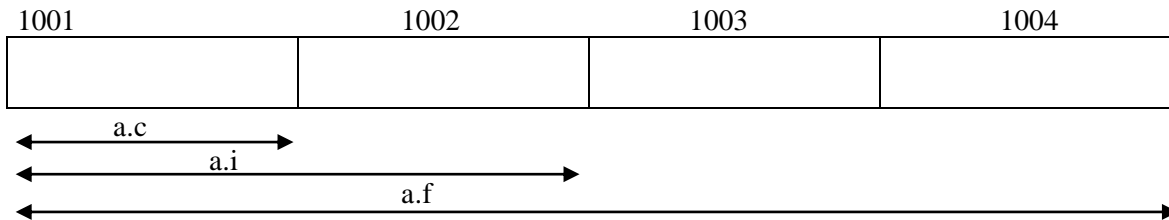union tag_name var1, var2...;
```

A union definition and variable declaration can be done by using any one of the following

union u	union u	typedef union
{	{	{
char c;	char c;	char c;
int i;	int i;	int i;
float f;	float f;	float f;
};	} a;	}U;
union u a;		U a;

a) Types of unions definitions

We can access various members of the union as mentioned: a.ca.ia.f and memory organization is shown below,

b) Memory Organization union

- In the above declaration, the member f requires 4 bytes which is the largest among all the members.
- Figure shows how all the three variables share the same address. The size of the union here is 4 bytes.
- A union creates a storage location that can be used by any one of its members at a time.
- When a different member is assigned a new value, the new value supersedes the previous members' value.

POINTERS

DEFINITION:

➢ Pointer is a user defined data type that creates special types of variables which can hold the address of primitive data type like char, int, float, double or user defined data type like function, pointer etc. or derived data type like array, structure, union, enum.

Examples:

> **int *ptr;**

➢ In c programming every variable keeps two types of value.
1. Value of variable.
2. Address of variable where it has stored in the memory.

(1) Meaning of following simple pointer declaration and definition:

```
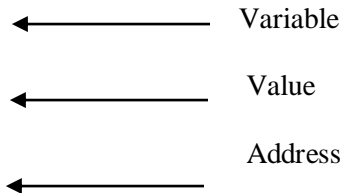int a=5;
int* ptr;
ptr=&a;
```

Explanation:

About variable ―a‖ :
1. Name of variable: a
2. Value of variable which it keeps: 5
3. Address where it has stored in memory: 1025 (assume)

About variable ―ptr‖ :
4. Name of variable: ptr
5. Value of variable which it keeps: 1025
6. Address where it has stored in memory: 5000 (assume)

Pictorial representation:

Variable

Note: A variable where it will be stored in memory is decided by operating system. We cannot guess at which location a particular variable will be stored in memory.

Value

Address

Pointers are built on three underlying concepts which are illustrated below:-

➢ Memory addresses within a computer are referred to as *pointer constants*. We cannot change them. We can only use them to store data values. They are like house numbers.

➢ We cannot save the value of a memory address directly. We can only obtain the value through the variable stored there using the address operator (&). The value thus obtained is known as *pointer value*. The pointer value (i.e. the address of a variable) may change from one run of the program to another.

➢ Once we have a pointer value, it can be stored into another variable. The variable that contains a pointer value is called a *pointer variable.*

| Pointer Constants | Pointer Values | Pointer Variables |

Pointe rs

Benefits of using pointers are:-
1) Pointers are more efficient in handling arrays and data tables.
2) Pointers can be used to return multiple values from a function via function arguments.
3) The use of pointer arrays to character strings results in saving of data storage space in memory.
4) Pointers allow C to support dynamic memory management.
5) Pointers provide an efficient tool for manipulating dynamic data structures such as structures , linked lists , queues , stacks and trees.
6) Pointers reduce length and complexity of programs.
7) They increase the execution speed and thus reduce the program execution time.

DECLARATION OF A POINTER VARIABLE:
➢ The declaration of a pointer variable takes the following form:

data_type *pt_name;

This tells the compiler three things about the variable pt_name:
1) The * tells that the variable pt_name is a pointer variable
2) pt_name needs a memory location
3) pt_name points to a variable of type data_type

Example: int *p;

Declares the variable p as a pointer variable that points to an integer data type.

ptr		a
1025	➜	5
5000		1025

INITIALIZATION OF POINTER VARIABLES:

➢ The process of assigning the address of a variable to a pointer variable is known as *initialization.*
➢ Once apointer variable has been declared we can use assignment operator to initialize the variable.

Example:

```
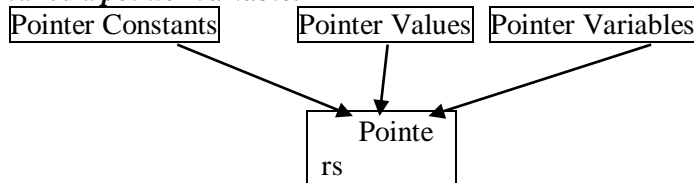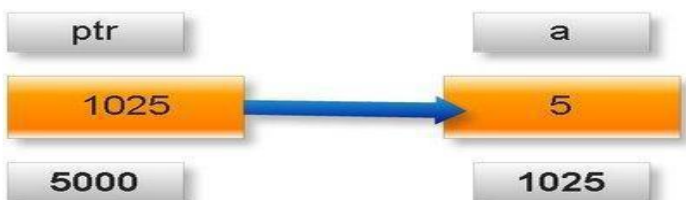int quantity ;
int *p; //declaration
p=&quantity; //initialization
```

➢ We can also combine the initialization with the declaration:

```
int *p=&quantity;
```

➢ Always ensure that a pointer variable points to the corresponding type of data.
➢ It is also possible to combine the declaration of data variable, the declaration of pointer variable and the initialization of the pointer variable in one step.

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

ACCESSING THE ADDRESS OF A VARIABLE:

➢ The actual location of a variable in the memory is system dependent and therefore the address of a variable is not known to us immediately.
➢ In order to determine the address of a variable we use & operator in c. This is also called as the **address operator.**
➢ The operator & immediately preceding a variable returns the address of the variable associated with it.

Syntax: p=&x;

would assign the address 5000 to the variable p. The & operator can be remembered as **address of.**

Example program:

```
main( )
{
int a = 5 ;
printf( "\nAddress of a = %u", &a );
printf( "\nValue of a = %d", a );
}
```

Output:

Address of a=1444
Value of a=5

➢ The expression &a returns the address of the variable a, which in this case happens to be 1444.
➢ Hence it is printed out using %u, which is a format specified for printing an unsigned integer.

ACCESSING A VARIABLE THROUGH ITS POINTER:

➢ Once a pointer has been assigned the address of a variable, to access the value of the variable using pointer we use the operator *'*', called 'value at address' operator*.
➢ It gives the value stored at a particular address. The **"value at address"** operator is also called *'indirection'* **operator (or dereferencing operator)**.

Some statement:

```
int quantity, *p, n;
quantity = 190;
p = &quantity;
n = *p;
```

CHAIN OF POINTER:

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

> A variable that is a pointer to a pointer must be declared as such. This is done by placing an additional asterisk in front of its name.
> For example, the following declaration declares a pointer to a pointer of type int –

```
int   **var;
```

When a target value is indirectly pointed to by a pointer to a pointer, accessing that value requires that the asterisk operator be applied twice, as is shown below in the example –

```c
#include<stdio.h>

int main ()
{
intvar;
int*ptr;
int**pptr;

var=3000;

/* take the address of var */
ptr=&var;

/* take the address of ptr using address of operator & */
pptr=&ptr;

/* take the value using pptr */
printf("Value of var = %d\n",var);
printf("Value available at *ptr = %d\n",*ptr);
printf("Value available at **pptr = %d\n",**pptr);

return0;
}
```

OUTPUT:
Value of var = 3000
Value available at *ptr = 3000
Value available at **pptr = 3000

POINTERS EXPRESSIONS:
> We can use pointer variables in expression.

Example:

> **int x = 10, y = 20, z;**
> **int *ptr1 = &x;**
> **int *ptr2 = &y;**
> **z = *ptr1 * *ptr2 ;**
> **Will assign 200 to**
> **variable z.**

➢ We can perform addition and subtraction of integer constant from pointer variable.
Example:

> **ptr1 = ptr1 + 2;**
> **ptr2 = ptr2 – 2;**

➢ We can not perform addition, multiplication and division operations on two pointer variables.
Example:

> ptr1 + ptr2   is not valid

➢ However we can subtract one pointer variable from another pointer variable. We can use increment and decrement operator along with pointer variable to increment or decrement the address contained in pointer variable.
Example:

> **ptr1++;**
> **ptr2--;**

➢ We can use relational operators to compare pointer variables if both pointer variable points to the variables of same data type.

POINTER INCREMENT AND SCALE FACTOR

➢ We can use increment operator to increment the address of the pointer variable so that it points to next memory location.
➢ The value by which the address of the pointer variable will increment is not fixed. It depends upon the data type of the pointer variable.
For Example:

> **int *ptr;**
> **ptr++;**

➢ **It will increment the address of pointer variable by 2. So if the address of pointer variable is 2000 then after increment it becomes 2002.**
➢ Thus the value by which address of the pointer variable increments is known as scale factor. The scale factor is different for different data types as shown below:

Char	1 Byte
Int	2 Byte
Short int	2 Byte
Long int	4 Byte
Float	4 Byte

| Double | 8 Byte |
| Long double | 10 Byte |

POINTERS AND ARRAYS:

- ➢ When an array is declared the complier allocates a base address and sufficient amount of storage to contain all the elements of the array in contiguous memory locations.
- ➢ The base address is the location of the first element (index 0) of the array .
- ➢ The complier also defines the array name as a constant pointer to the first element.

        Example :-   **static int x[5]= {1,2,3,4,5};**

- ➢ Suppose the base address of x is 1000 and assuming that each integer requires two bytes.
- ➢ The five elements will be stored as follows.

elements ⟶ x[0]   x[1] x[2]   x[3]   x[4]

value ⟶

1	2	3	4	5

address 1000 ⟶   1002   1004   1006   1008

                        Base address

- ➢ The name x is defined as a constant pointer pointing to the first element, x[0], and therefore the value x is 1000, the location where x[0] is stored.
- ➢ That is   **x=&x[0]=1000;**
- ➢ If we declare p as an integer pointer, then we can make the pointer p to the array x by the following assignment

                p=x;

        which is equivalent to p=&x[0];

- ➢ Now we can access every value of x using p++ to move from one element to another.
- ➢ The relationship between p and x is shown below

             p=&x[0]=1000
             p+1=&x[1]=1002
             p+2=&x[2]=1004
             p+3=&x[3]=1006
             p+4=&x[4]=1008

Note:- address of an element in an array is calculated by its index and scale factor of the datatype.
Address of x[n] = base address + (n*scale factor of type of x).

        Eg:-   **int x[5]; x=1000;**
Address of x[3]= base address of x+ (3*scale factor of int)

        = 1000+(3*2)
        = 1000+6
        =1006

        Eg:- float avg[20]; avg=2000;
Address of avg [6] =2000+(6*scale factor of float)
               =2000+6*4 =2000+24=2024.

        Eg:- char str [20]; str =2050;
Address of str[10]=2050+(10*1)
             =2050+10
             =2060.

Note2:- when handling arrays, of using array indexing we can use pointers to access elements.
Like *(p+3) given the value of x[3]. The pointer accessing method is faster than the array indexing.

POINTERS AND CHARCTERS STRINGS:

> String is an array of characters terminated with a null character. We can also use pointer to access the individual characters in a string .this is illustrated as below

Note :- In `c` a constant character string always represents a pointer to that string and the following statement is valid

<p align="center">char *name;
name = "delhi";</p>

these statements will declare name as a pointer to a character and assign to name the constant character string "Delhi".

This type of declarations is not valid for character string .

Like:-

```
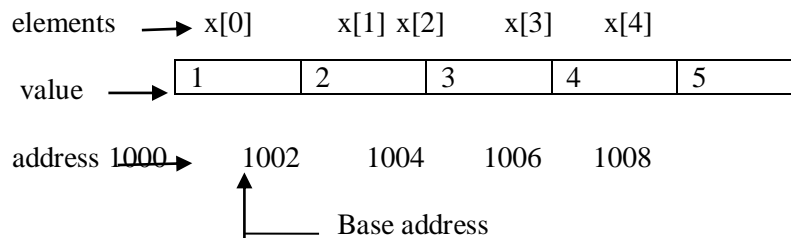            char name [20];
            name ="delhi" ;           //invalid
```

ARRAY OF POINTER:

> We have studied array of different primitive data types such as int, float, char etc. Similarly C supports array of pointers i.e. collection of addresses.

Example :-

```
                void main( )
                {
                int *ap[3];
                int al[3]={10,20,30};
                int k;
                for(k=0;k<3;k++)
            ap[k]=al+k;
            printf("\n address element\n");
            for(k=0;k<3;k++)
            {
            printf("\t %u",ap[k]);
            printf("\t %7d\n",*(ap[k]));
            }
            }
```

Output:		
Address		**Element**
4060	**10**	
4062		**20**
4064		**30**

> In the above program , the addresses of elements are stored in an array and thus it represents array of pointers.
> A two-dimensional array can be represented using pointer to an array. But, a two-dimensional array can be expressed in terms of array of pointers also. The conventional array definition is,

data_type array name [exp1] [exp2];

> Using array of pointers, a two-dimensional array can be defined as,

data_type *array_name [exp1];

Where,

- data_type refers to the data type of the array.
- array_name is the name of the array.
- exp1 is the maximum number of elements in the row.

Note that exp2 is not used while defining array of pointers. Consider a two-dimensional vector initialized with 3 rows and 4 columns as shown below,

int p[3][4]={{10,20,30,40},{50,60,70,80},{25,35,45,55}};

The elements of the matrix p are stored in memory row-wise (can be stored column-wise also).

> Using array of pointers we can declare p as,

int *p[3];

Here, p is an array of pointers. p[0] gives the address of the first row, p[1] gives the address of the second

row and p[2] gives the address of the third row.

Now, p[0]+0 gives the address of the element in 0th row and 0th column, p[0]+1 gives the address of the elements in 0th row and 1st column and so on.

In general,

        - Address of ith row is given by a[i].

        - Address of an item in ith row and jth column is given by, p[i]+j.

        - The element in ithrow and jth column can be accessed using the indirection operator * by specifying, *(p[i]+j).

POINTER AS FUNCTION ARGUMENTS:

- ➢ When we pass addresses to a function, the parameter receiving the addresses should be pointers.
- ➢ The process of calling a function using pointers to pass the address of variables is known as "call by reference".
- ➢ The function which is called by reference can change the value of the variable used in the call.

  Example :-

```
void main()
{
int x;
x=50;
change(&x); /* call by reference or address */
printf("%d\n",x);
}
change change(int *p)
{
*p=*p+10;
}
```

- ➢ When the function change ( ) is called, the address of the variable x, not its value, is passed into

the function change ( ). Inside change (), the variable **p** is declared as a pointer and therefore **p** is the address of the variable x. The statement,

$$*p = *p + 10;$$

Means —add 10 to the value stored at the address **p**‖. Since **p** represents the address of **x**, the value of x is changed from 20 to 30. Thus the output of the program will be 30, not 20.Thus, call by reference provides a mechanism by which the function can change the stored values in the calling function.

FUNCTION RETURNING POINTER:

- ➢ Functions return multiple values using pointers. The return type of function can be a pointer of type int , float ,char etc.

Example :

```
#include<stdio.h>
int * smallest(int * , int *);
void main()
{
int a,b,*s;
printf("Enter a,b values ");
scanf("%d%d",&a,&b);
s=smallest(&a,&b);
printf("smallest no. is %d",*s);
}
int * smallest(int *a, int *b)
{
```

```
if(*a<*b)
return a;
else
return b;
}
```

In this example, "return a" implies the address of "a" is returned, not value .So in order to hold the address, the function return type should be pointer.

POINTERS TO FUNCTIONS

- ➢ A function, like a variable has a type and address location in the memory.
- ➢ It is therefore possible to declare a pointer to a function, which can then be used as an argument in another function.   A pointer to a function can be declared as follows.

type (*fptr)();

- ➢ This tells the complier that fptr is a pointer to a function which returns type value the parentheses around *fptr is necessary.
- ➢ Because type *gptr(); would declare gptr as a function returning a pointer to type.
- ➢ We can make a function pointer to point to a specific function by simply assigning the name of the function to the pointer.

Example:

double mul(int,int);
double (*p1)();
p1=mul();

- ➢ It declares p1 as a pointer to a function and mul as a function and then make p1 to point to the function mul.
- ➢ To call the function mul we now use the pointer p1 with the list of parameters.

i,e (*p1)(x,y); //function call equivalent to mul(x,y);

Program:

```
double mul(int ,int);
void main()
{
int x,y;
double (*p1)();
double res;
p1=mul;
printf("\n enter two numbers:");
scanf("%d %d",&x,&y);
res=(*p1)(x,y);
printf("\n The Product of X=%d and Y=%d is res=%lf",x,y,res);
}
double mul(int a,int b)
{
double val;
val=a*b;
return(val);
}
```

Output:

```
using pointers to function
 enter two numbers:22 7

The Product of X=22 and Y=7 is res=154.000000_
```

POINTERS AND STRUCTURES:
- A pointer can also point to a structure.

Example :
```
struct student
{
        int sno;
char sname[20], course[20];
float fee;
   };
struct student s;
struct student *p;
```

Here p is defined to be a pointer, pointing to student structure, we can write

$$p = \&s;$$

After making such an assignment we can access every data item of student structure through **p .**
è(Arrow)
_____

-> is the combination of – followed by > .
It is used to access the data items of a structure with the help of structure pointer.

syntax:

Struct_pointer -> data item;

Eg:

pàsno;
pàsname;
pà course;
pà fee;

FILE MANAGEMENT IN C

- File management in C, File operation functions in C, Defining and opening a file, Closing a file, The getw and putw functions, The fprintf&fscanf functions, Random access to files and fseek function.
- C supports a number of functions that have the ability to perform basic file operations, which include:
  1. Naming a file
  2. Opening a file
  3. Reading from a file
  4. Writing data into a file
  5. Closing a file

- Real life situations involve large volume of data and in such cases, the console oriented I/O   operations pose two major problems
- It becomes cumbersome and time consuming to handle large volumes of data through terminals.
- The entire data is lost when either the program is terminated or computer is turned off therefore it is necessary to have more flexible approach where data can be stored on the disks and read whenever necessary, without destroying the data. This method employs the concept of files to store data.

File operation functions in C:

Function Name	Operation
fopen()	Creates a new file for use Opens a new 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
fprintf()	Writes a set of data values to a file
fscanf()	Reads a set of data values from a file
getw()	Reads a integer from a file
putw()	Writes an integer to the file
fseek()	Sets the position to a desired point in the file
ftell()	Gives the current position in the file
rewind()	Sets the position to the begining of the file

DEFINING AND OPENING A FILE:

- ➢ If we want to store data in a file into the secondary memory, we must specify certain things about the file to the operating system.
- ➢ They include the
    1. **filename,**
    2. **data structure,**
    3. **purpose**.
- ➢ The general format of the function used for opening a file is

    FILE *fp;

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

The first statement declares the variable fp as a pointer to the data type FILE. As stated earlier, File is a structure that is defined in the I/O Library.

The second statement opens the file named filename and assigns an identifier to the FILE type pointer fp. This pointer, which contains all the information about the file, is subsequently used as a communication link between the system and the program.

The second statement also specifies the purpose of opening the file. The mode does this job.

        r   **open the file for read only.**

        w   **open the file for writing only.**

a   **open the file for appending data to it.**

Consider the following statements:

        FILE *p1, *p2;

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

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

In these statements the p1 and p2 are created and assigned to open the files data and results respectively the file data is opened for reading and result is opened for writing. In case the results file already exists, its contents are deleted and the files are opened as a new file. If data file does not exist error will occur.

CLOSING A FILE:

➢ The input output library supports the function to close a file; it is in the following format.
fclose(file_pointer);
➢ A file must be closed as soon as all operations on it have been completed.
➢ This would close the file associated with the file pointer.
   Observe the following program.

....
FILE *p1 *p2;
p1=fopen ("Input","w");
p2=fopen ("Output","r");
....
...
fclose(p1);
fclose(p2)

The above program opens two files and closes them after all operations on them are completed, once a file is closed its file pointer can be reversed on other file.

INPUT /OUTPUT OPERATIONS ON FILES:

The getc and putc functions:
• The getc and putc functions are analogous to getchar and putchar functions and handle one character at a time.
• The putc function writes the character contained in character variable c to the file associated with the pointer fp1.
   Example: **putc(c,fp1);**
• The getc function is used to read a character from a file that has been open in read mode.
   Example: **c=getc(fp2);**
• The program shown below displays use of a file operations. The data enter through the keyboard and the program writes it. Character by character, to the file input.
• The end of the data is indicated by entering an EOF character, which is control-z. the file input is closed at this signal.

```
#include< stdio.h>
main()
{
file *f1;
printf("Data input output");
f1=fopen("Input","w"); /*Open the file Input*/
while((c=getchar())!=EOF) /*get a character from key board*/
putc(c,f1); /*write a character to input*/
fclose(f1); /*close the file input*/
printf("nDataoutputn");
```

```
f1=fopen("INPUT","r"); /*Reopen the file input*/
while((c=getc(f1))!=EOF)
printf("%c",c);
fclose(f1);
}
```

The getw and putw functions:

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

> **putw(integer,fp);**
> **getw(fp);**

```
/*Example program for using getw and putw functions*/
#include<stdio.h>
main()
{
FILE *f1,*f2,*f3;
int number I;
printf("Contents of the data filenn");
f1=fopen("DATA","W");
for(I=1;I< 30;I++)
{
scanf("%d",&number);
if(number==-1)
break;
putw(number,f1);
}
fclose(f1);
f1=fopen("DATA","r");
f2=fopen("ODD","w");
f3=fopen("EVEN","w");
while((number=getw(f1))!=EOF)/* Read from data file*/
{
if(number%2==0)
putw(number,f3);/*Write to even file*/
else
putw(number,f2);/*write to odd file*/
}
fclose(f1);
fclose(f2);
fclose(f3);
f2=fopen("ODD","r");
f3=fopen("EVEN","r");
```

```
printf("nnContents of the odd filenn");
while(number=getw(f2))!=EOF)
printf("%d%d",number);
printf("nnContents of the even file");
while(number=getw(f3))!=EOF)
printf("%d",number);
fclose(f2);
fclose(f3);
}
```

The fprintf&fscanf functions:

- The fprintf and fscanf functions are identical to printf and scanf functions except that they work on files. The first argument of these functions is a file pointer which specifies the file to be used.
The general format of fprintf:

> **fprintf(fp,"control string", list);**

- Where fp id a file pointer associated with a file that has been opened for writing. The control string is file output specifications list may include variable, constant and string.

> **fprintf(f1,%s%d%f",name,age,7.5);**

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

The general format of fscanf:

> **fscanf(fp,"controlstring",list);**

This statement would cause the reading of items in the control string.

Example: **fscanf(f2,"5s%d",item,&quantity");**

Like scanf, fscanf also returns the number of items that are successfully read.

```
/*Program to handle mixed data types*/
#include<stdio.h>
main()
{
FILE *fp;
int num,qty,I;
float price,value;
char item[10],filename[10];
printf("Input filename");
scanf("%s",filename);
fp=fopen(filename,"w");
printf("Input inventory datann"0;
printf("Item namem number price quantityn");
for I=1;I< =3;I++)
{
```

```
fscanf(stdin,"%s%d%f%d",item,&number,&price,&quality);
fprintf(fp,"%s%d%f%d",itemnumber,price,quality);
}
fclose (fp);
fprintf(stdout,"nn");
fp=fopen(filename,"r");
printf("Item name number price quantity value");
for(I=1;I< =3;I++)
{
fscanf(fp,"%s%d%f%d",item,&number,&prince,&quality);
value=price*quantity");
fprintf("stdout,"%s%d%f%d%dn",item,number,price,quantity,value);
}
fclose(fp);
}
```

RANDOM ACCESS TO FILES:

- There is no need to read each record sequentially, if we want to access a particular record.C supports these functions for random access file processing.

  1. **fseek( )**
  2. **ftell( )**
  3. **rewind( )**

The ftell Function:

- The `ftell()` function tells us about the current position in the file (in bytes).

  Syntax:

  > **pos = ftell(fptr);**

  Where, **fptr** is a file pointer. **pos** holds the current position i.e., total bytes read (or written).

  Example:

  If a file has 10 bytes of data and if the **ftell( )** function returns 4 then, it means that 4 bytes has already been read (or written).

The rewind Function:

- We use the `rewind()` function to return back to the starting point in the file.

  Syntax:

  > **rewind(fptr);**

  Where, **fptr** is a file pointer.

The fseek Function:

- We use **the `fseek( )`** function to move the file position to a desired location.

Syntax:

```
fseek(fptr, offset, position);
```

Where, `fptr` is the file pointer. `offset` which is of type `long`, specifies the number of positions (in bytes) to move in the file from the location specified by the `position`.

The `position` can take the following values.

- 0 - The beginning of the file
- 1 - The current position in the file
- 2 - End of the file

Following are the list of operations we can perform using the `fseek()` function.

Operation	Description
`fseek(fptr, 0, 0)`	This will take us to the beginning of the file.
`fseek(fptr, 0, 2)`	This will take us to the end of the file.
`fseek(fptr, N, 0)`	This will take us to (N + 1)th bytes in the file.
`fseek(fptr, N, 1)`	This will take us N bytes forward from the current position in the file.
`fseek(fptr, -N, 1)`	This will take us N bytes backward from the current position in the file.
`fseek(fptr, -N, 2)`	This will take us N bytes backward from the end position in the file.

ERROR HANDLING IN FILES
- ➤ 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 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 operations

.

5. Opening a file with an invalid filename.

6. Attempting to write a write protected file.

➢ If we fail to check such read and write errors, a program may behave abnormally when an error occurs.

➢ An unchecked error may result in a premature termination of the program or incorrect output.

➢ In C we have two status - inquiry library functions **feof and ferror**that can help us detect I/O errors in the files.

a). feof():
- The feof() function can be used to test for an end of file condition.
- It takes a FILE pointer as its only argument and returns a non zero integer value if all of the data from the specified file has been read, and returns zero otherwise.
- If fp is a pointer to file that has just opened for reading, then the statement
    if(feof(fp))
    printf("End of data");
would display the message "End of data" on reaching the end of file condition.

b).ferror():
- The ferror() function reports the status of the file indicated.
- It also takes a file pointer as its argument and returns a nonzero integer if an error has been detected up to that point, during processing.
- It returns zero otherwise. The statement
    if(ferror(fp)!=0)
    printf("an error has occurred\n");
- would print an error message if the reading is not successful.

c). fp==null:
- We know that whenever a file is opened using fopen function, a file pointer is returned.
- If the file cannot be opened for some reason, then the function returns a null pointer.
- This facility can be used to test whether a file has been opened or not.
    Example
    if(fp==NULL)
    printf("File could not be opened.\n");

d) perror( ):
- It is a standard library function which prints the error messages specified by the compiler.
    Example:
    if(ferror(fp))
    perror(filename);

Program for error handling in files:
#include<stdio.h>
void main( )
{ FILE *fp;
char ch;
fp=fopen("my1.txt","r");
if(fp==NULL)

```
while(!feof(fp))
{
ch=getc(fp);
if(ferror(fp))
perror("problem in the file");
else
printf("\n file cannot be opened");
putchar(ch);
}
fclose(fp);
 }
```

COMMAND LINE ARGUMENT:

- ➤ It is the parameter supplied to a program when the program is invoked.
- ➤ This parameter may represent a file name the program should process.
- ➤ For example, if we want to execute a program to copy the contents of a file named X_FILE to another one name Y_FILE then we may use a command line like

C:> program X_FILE Y_FILE

- ➤ Program is the file name where the executable code of the program is stored.
- ➤ This eliminates the need for the program to request the user to enter the file names during execution.
- ➤ The "main" function can take two arguments called argc, argv and information contained in the command line is passed on to the program to these arguments, when "main" is called up by the system.
- ➤ The variable **argv**is an argument vector and represents an array of characters pointers that point to the command line arguments.
- ➤ The **argc**is an argument counter that counts the number of arguments on the command line.
- ➤ The size of this array is equal to the value of argc. In order to access the command line arguments, we must declare the "main" function and its parameters as follows:

main(argc,argv)
int argc;
char *argv[ ];
{
……….
}
Generally argv[0] represents the program name.

UNIT-IV COMPLETED

DYNAMIC MEMORY ALLOCATION

DEFINITION:
- ➤ The process of allocating memory at runtime is known as **dynamic memory allocation**.
- ➤ Library routines known as **memory management functions** are used for allocating and freeing memory during execution of a program. These functions are defined in **stdlib.h** header file.

Function	Description
`malloc()`	allocates requested size of bytes and returns a void pointer pointing to the first byte of the allocated space
`calloc()`	allocates space for an array of elements, initialize them to zero and then returns a void pointer to the memory
`Free`	releases previously allocated memory
`Realloc`	modify the size of previously allocated space

Memory Allocation Process:
- ➤ **Global** variables, `static` variables and program instructions get their memory in **permanent** storage area whereas **local** variables are stored in a memory area called **Stack**.
- ➤ The memory space between these two region is known as **Heap** area. This region is used for dynamic memory allocation during execution of the program. The size of heap keep changing.
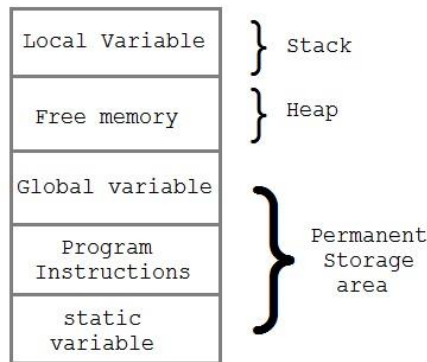
Allocating block of Memory:
i) malloc( ) function :
- • It is used for allocating block of memory at runtime.
- • This function reserves a block of memory of the given size and returns a **pointer** of type `void`.
- • This means that we can assign it to any type of pointer using typecasting.
- • If it fails to allocate enough space as specified, it returns a `NULL`pointer.
    - **Syntax:**

> void*malloc(byte-size)

Example: malloc()

> int *x;
> x =(int*)malloc(50*sizeof(int));//memory space allocated to variable
> x
> free(x);//releases the memory allocated to variable x

ii) calloc( ) function: **(allocates multiple blocks of memory)**

- It is another memory allocation function that is used for allocating memory at runtime.
- calloc function is normally used for allocating memory to derived data types such as **arrays** and **structures**.
- If it fails to allocate enough space as specified, it returns a NULL pointer.
  Syntax:

> void *calloc(number of items, element-size)

Example: calloc( )

> **struct employee**
> **{**
>   **char *name;**
>   **int salary;**
> **};**
> **typedef struct employee emp;**
> **emp *e1;**
> **e1**
> **=(emp*)calloc(30,sizeof(emp));**

iii) realloc( ) **function: (grows or shrinks allocated memory)**

- It changes memory size that is already allocated dynamically to a variable.
  Syntax:

> void*realloc(pointer,new-size)

Example: realloc()

> **int *x;**
> **x =(int*)malloc(50*sizeof(int));**
> **x =(int*)realloc(x,100);//allocated a new memory to variable x**

iv) free( ) function: (de-allocates memory)

- The function free( ) has the following prototype:

> **free(ptr);**

- The function free( ) de-allocates a memory block pointed by **ptr.**

- **ptr** is the pointer that is points to allocated memory by malloc( ), calloc( ) or realloc().
- Passing an uninitialized pointer, or a pointer to a variable not allocated by malloc( ), calloc() or realloc()could be dangerous and disastrous.

   Example:

   ```
   int *a;
   a=(int *) malloc(30); //first 30 bytes of memory is allocated.
   free(a); //de-allocates 30 bytes of memory.
   ```

Diffrence between malloc() and calloc()

calloc()	malloc()
calloc() initializes the allocated memory with 0 value.	malloc() initializes the allocated memory with garbage values.
Number of arguments is 2	Number of argument is 1
Syntax:(cast_type*)calloc(blocks, size_of_block);	**Syntax :** (cast_type *)malloc(Size_in_bytes);

LINKED LISTS

➤ A **linked list** is a way to store a collection of elements. Like an array these can be character or integers. Each element in a linked list is stored in the form of a **node**.

   Node:

➤ A node is a collection of two sub-elements or parts. A **data** part that stores the element and a **next** part that stores the link to the next node.

Linked List:

A linked list is formed when many such nodes are linked together to form a chain. Each node points to the next node present in the order. The first node is always used as a reference to traverse the list and is called **HEAD**. The last node points to **NULL**.

Declaring a Linked list :

In C language, a linked list can be implemented using structure and pointers

ADVANTAGES OF LINKED LISTS:

   i.   Linked lists are dynamic data structures. i.e., they can grow or shrink during the execution of a program.

ii. Linked lists have efficient memory utilization. Memory is allocated whenever it is required and it is de-allocated (removed) when it is no longer needed.

iii. Insertion and Deletions are easier and efficient. Linked lists provide flexibility in inserting a data item at a specified position and deletion of the data item from the given position.

iv. Many complex applications can be easily carried out with linked lists.

TYPES OF LINKED LISTS:

➤ There are 4 different implementations of Linked List available, they are:

1. Singly Linked List
2. Doubly Linked List
3. Circular Linked List
4. Doubly Circular Linked List

Singly Linked List:

- Singly linked lists contain nodes which have a **data** part as well as an **address part** i.e. `next`, which points to the next node in the sequence of nodes.
- The operations we can perform on singly linked lists are **insertion**, **deletion** and **traversal**.
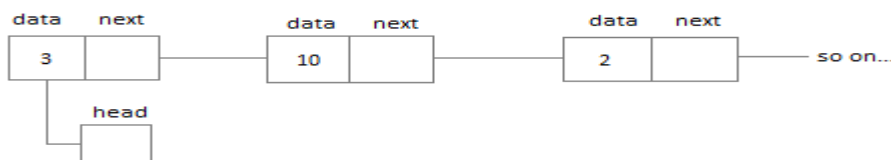
Doubly Linked List:

➤ In a doubly linked list, each node contains a **data** part and two addresses, one for the **previous** node and one for the **next** node.
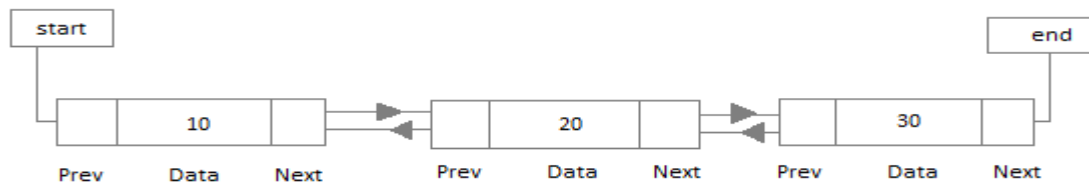
Circular Linked List:

➤ In circular linked list the last node of the list holds the address of the first node hence forming a circular chain.
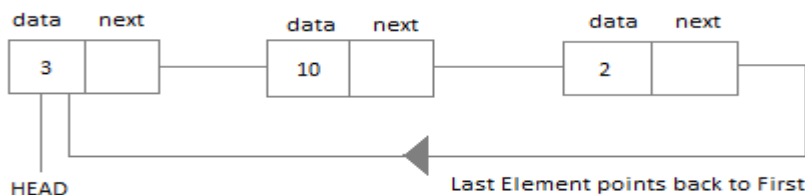
Doubly Circular Linked List:

- Doubly circular linked list is a linked data structure which consists of a set of sequentially linked records called nodes.

- Doubly circular linked list can be conceptualized as two singly linked lists formed from the same data items, but in opposite sequential orders.
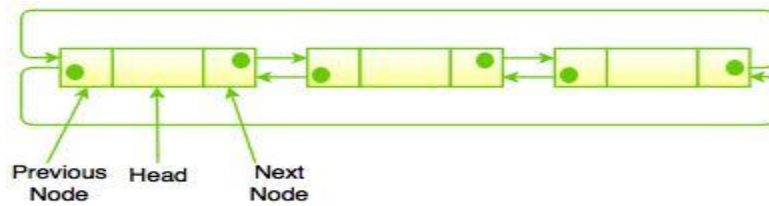
Fig. Doubly Circular Linked List

THE PREPROCESSOR

➤ The **C Preprocessor** is not a part of the compiler, but is a separate step in the compilation process.
➤ In simple terms, a C Preprocessor is just a text substitution tool and it instructs the compiler to do required pre-processing before the actual compilation. We'll refer to the C Preprocessor as CPP.
➤ All preprocessor commands begin with a hash symbol (#).
➤ It must be the first nonblank character, and for readability, a preprocessor directive should begin in the first column.
➤ The following section lists down all the important preprocessor directives −

Sr.No.	Directive & Description
1	**#define** Substitutes a preprocessor macro.
2	**#include** Inserts a particular header from another file.
3	**#undef** Undefines a preprocessor macro.
4	**#ifdef** Returns true if this macro is defined.
5	**#ifndef** Returns true if this macro is not defined.
6	**#if** Tests if a compile time condition is true.
7	**#else**The alternative for #if.
8	**#elif** #else and #if in one statement.
9	**#endif** Ends preprocessor conditional.
10	**#error** Prints error message on stderr.
11	**#pragma** Issues special commands to the compiler, using a standardized method.

> ➢ These directives can be classified into three categories:
>   1. Macro Substitution
>   2. File Inclusion Directives.
>   3. Compiler Control Directives.

MACRO SUBSTITUTION:
> ➢ It is a process where an identifier in a program is replaced by a predefined string composed of one or more tokens.
> ➢ The preprocessor accomplishes this task under the direction of **#define** statement.
>   **Syntax:**

>> **#define identifier string**

> • There are different forms of macro substitutions. The most common forms are:
>   1. Simple macro substitutions.
>   2. Argumented macro substitutions.
>   3. Nested macro substitutions.

1) Simple macro substitutions:
>   o Simple string replacement is commonly used to define constants.
>   **Example:**

>> **#define   COUNT   100**
>> **#define   FALSE   0**
>> **#define   PI   3.14**
>> **#define   CAPITAL   " DELHI"**

> **NOTE:** That we have written all macros(identifiers) in capitals.
>   o It is a convention to write all macros in capitals to identify them as symbolic constants.

>> **#define   M   5**

> Will replace all occurrences of M with 5, starting from the line of definition to the end of the program. However, a macro inside a string does not get replaced. Consider the following two lines:

>> **Total = M* value**
>> **printf("M = %d \n" , M);**

2) Macro with Arguments:
>   o The preprocessor permits us to define more complex and more useful form of replacements.
>   **Syntax:**

>> **#define   identifier(f1,f2,…….,fn)   string**

>   **Example:**

>> **#define   CUBE(X)   (X*X*X)**

>   If the following statement appears later in the program
>>   **Volume= CUBE(side);**

> **Is equal to ,**

>>   **Volume = (side * side * side);**

> **Another Example,**

>> **#define   MAX(a,b)   (((a)>(b))?(a) : (b))**
>> **#define   MIN(a,b)   (((a)<(b))?(a) : (b))**

3) **Nesting of Macros:**
- o It can use also one macro in the definition of another macro. That is, macro definitions may be nested.
- o **Example:**

                    #define M   5
                    #define   N   M+1
                    #define   SQUARE(x)   ((x)*(x))
                    #define CUBE(x)   (SQUARE (x) *(x))

Undefining a Macro:
- o A defined macro can be undefined, using the statement
         #undef   identifier
- o This is useful when we want to restrict the definition only to a particular part of the program.

FILE INCLUSION:

- ➢ File inclusive Directories are **used to include user define header file**inside C Program.
- ➢ File inclusive directory checks included header file inside same directory (if path is not mentioned).
- ➢ File inclusive directives begins with **#include**
- ➢ If Path is mentioned then it will include that **header file into current scope**.
- ➢ Instead of using triangular brackets we use "**Double Quote**" for inclusion of user defined header file.
- ➢ It instructs the compiler to include **all specified files.**

Ways of including header file

way 1 : Including Standard Header Files

```
#include<filename>
```

- • Search File in Standard Library

way 2 :User Defined Header Files are written in this way

```
#include"FILENAME"
```

- • Search File in Current Library
- • If not found , Search File in Standard Library
- • User defined header files are written in this format

Example :

```
#include<stdio.h>     // Standard Header File

#include<conio.h>     // Standard Header File

#include"myfunc.h"    // User Defined Header File
```

Explanation :

1. In order to include user defined header file inside C Program , we must have to **create one user defined header file**. [Learn How to Create User Defined Header File]
2. Using double quotes include user defined header file inside Current C Program.

3. "myfunc.h" is user defined header file .
4. **We can combine all our user defined functions** inside header file and can include header file whenever require.

COMPILER CONTROL DIRECTIVE

The C Preprocessor offer a feature known as conditional compilation, which can be used to switch on or off a particular line or group of lines in a program.

This is achieved by the inserting **#ifdef** or **#endif**.

Conditional selection of code using **#ifdef**,**#endif**.The preprocessor has a conditional statement similar to' C 's if else.

It can be used to selectively include statements in a program. This is often used where two different computer types implement a feature in different ways. It allows the programmer to produce a program which will run on either type.

The keywords for conditional selection are; **#ifdef**, **#else and #endif**.
#ifdef: -takes a name as an argument, and returns true if the name has a current definition. The name may be defined using a **#define**, the -d option of the compiler.

#else:-is optional and ends the block beginning with **#ifdef**. It is used to create a 2 way optional selection.

#endif:-ends the block started by **#ifdef** or **#else**.

Where the **#ifdef** is true, statements between it and a following **#else** or **#endif** are included in the program. Where it is false, and there is a following **#else**, statements between the **#else** and the following **#endif** are included.

This is best illustrated by an example.Using **#ifdef** for Different Computer Types.Conditional selection is rarely performed using **#defined** values.

UNIT-V COMPLETED