

MODULE 1

INTRODUCTION TO C LANGUAGE

Syllabus

- ✓ **Pseudo code Solution to Problem**
- ✓ **Basic Concepts of C Program**
- ✓ **Declaration, Assignment & Print Statements**
- ✓ **Data Types**
- ✓ **Operators and Expressions**
- ✓ **Programming Examples and Exercise.**

1.1 Introduction to C

- C is a general-purpose programming language developed by **Dennis Ritchie** at AT&T Bell laboratories in 1972.

Advantages/Features of C Language

C language is very popular language because of the following features:

1. C is structured Programming Language
2. It is considered a high-level language because it allows the programmer to solve a problem without worrying about machine details.
3. It has wide variety of operators using which a program can be written easily to solve a given problem.
4. C is more efficient which increases the speed of execution and management of memory compared to low level languages.
5. C is machine independent. The program written on one machine will work on another machine.
6. C can be executed on many different hardware platforms.

1.2 Pseudocode: A solution to Problem

- **Definition:**

- It is a series of steps to solve a given problem written using a mixture of English and C language.
- It acts as a problem solving tool.
- It is the first step in writing a program.

- **Purpose:**
 - Is to express solution to a given problem using mixture of English language and c like code.
- **Advantage:**
 - Easy to write and understand
 - It is relatively easy to convert English description solution of small programs to C program.

Ex 1: Addition of two numbers

1. Get the numbers[a,b]
2. Compute addition [Sum= a + b]
3. Print the results [Sum]

Ex 2: Area of Circle

1. Get the radius[r]
2. Compute area[Area = 3.141*r*r]
3. Print the results [Area]

Disadvantage:

- It is very difficult to translate the solution of lengthy and complex problem in English to C

C Programming Concepts

- **Program:** A program is a group of instructions given by the programmer to perform a specific task.

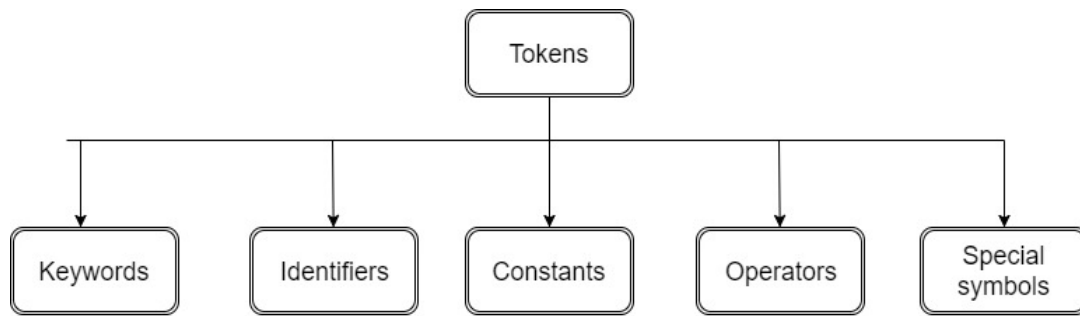
1.3 Character set of C language

- **Definition:** A symbol that is used while writing a program is called a character.
- A character can be:
 - ➔ **Alphabets/Letters (Lowercase a-z, Uppercase A-Z)**
 - ➔ **Digits (0-9)**
 - ➔ **Special Symbols (~ ‘ ! @ # % & * () - + / \$ = \ {**
 - ➔ **} [] : ; “ “ ? etc)**

Symbol	Name	Symbol	Name	Symbol	Name
~	Tilde		Vertical bar	[Left bracket
#	hash	(Left parenthesis]	Right bracket
\$	Dollar sign)	Right parenthesis	:	Colon
%	Percent sign	_	Underscore	"	Quotation mark
^	Caret	+	Plus sign	;	Semicolon
&	Ampersand	{	Left brace	<	Less than
*	Asterisk	}	Right brace	>	Greater than
'	Single quote	.	dot	=	Assignment
,	comma	\	backslash	/	Division

1.4 C Tokens

- Tokens are **the smallest or basic units of C program.**
- One or more characters are grouped in sequence to form meaningful words. these meaningful words are called as tokens.
- A token is **collection of characters.**
- Tokens are classified in to **5 types** as below:



1.4.1 Keywords

- The tokens which have predefined meaning in C language are called **keywords**.
- They are reserved for specific purpose in C language they are called as **Reserved Words**.
- There are totally **32 keywords** supported in C they are:

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

Rules for keywords

1. Keywords should **not be used as variables, function names, array names** etc.
2. All keywords should be written in **lowercase letters**.
3. Keywords **meaning cannot be changed by the users**.

1.4.2 Identifiers

Definition:

- Identifiers are the names given to program elements such as variables, constants, function names, array names etc
- It consists of one or more letters or digits or underscore.

Rules for identifiers

1. The First character should be an **alphabet** or an **underscore** _
Then First character is followed by any number of **letters or digits**.

2. No extra symbols are allowed other than **letters ,digits and Underscore**
3. Keywords cannot be used as an identifier
4. The length can be **31 characters** for external, **63** for internal.
5. Identifiers are **case sensitive**.

Example: Area, Sum_

Ex:-

Identifier	Reasons for invalidity
india06	Valid
_india	Valid
india_06	Valid
india_06_king	Valid
__india	valid
06india	not valid as it starts from digits
int	not valid since int is a keyword
india 06	not valid since there is space between india and 06
india@06	not valid since @ is not allowed

1.4.3 Constants

Definition:

- Constants refers to **fixed values** that **do not change** during the execution of a program
- The different types of constants are:
 1. **Integer constant**
 2. **Real constant/Floating Pointing constant**
 3. **Enumeration constant**
 4. **Character constant**
 5. **String constant**

1. Integer constant

- Definition: An integer is whole number without any decimal point.no extra characters are allowed other than + or _ .

- Three types of integers
- **Decimal integers:** are constants with a combination of digits 0 to 9, optional + or -
Ex: 123 , -345, 0 , 5436 , +79
- **Octal integers:** are constants with a combination of Digits from 0 to 7 but it has a prefix of 0
Ex: 027 , 0657 , 0777645
- **Hexadecimal integers:** Digits from 0 to 9 and characters from a to f, it has to start with 0X or 0x

Ex: 0X2 0x56 0X5fd 0xbdae

2. Real constants/Floating point:

- **Floating point** constants are base 10 numbers that are represented by fractional parts, such as 10.5. They can be positive or negative.
- Two forms are:

i. Fractional form:

- A floating point number represented using fractional form has an integer part followed by dot and a fractional part.
- Ex: 0.0083
- 215.5
- -71.
- +0.56 etc..

ii. Exponential form:

- A floating point number represented using **Exponent form has 3 parts**
- **mantissa e exponent**
- Mantissa is either real number expressed in decimal notation or integer.
- **Exponent must be integer with optional + or –**
- Ex 9.86 E 3 ⇒ 9.86×10^3
- Ex 9.86 E -3 ⇒ 9.86×10^{-3}

3. Enumeration constant

- A set of **named integer constants defined using the keyword enum** are called enumeration constants.
- Syntax:
enum identifier{enum list};

- `enum Boolean{NO,YES};`
NO is assigned with 0
YES is assigned with value 1
- `enum days{ mon,tue.wed};`
mon is assigned with 0
tue is assigned with 1
wed is assigned with 2

4.Character constant

- A symbol enclosed within pair of single quotes is called character constant.
- Each character is associated with unique value called ASCII value.
- Ex: '9'
- '\$'
- Backslash constants(Escape sequence character)
 - Definition: An escape sequence character begins with backslash and is followed by one character.
 - A backslash along with a character give rise to special print effects.
 - It always start with backslash ,hence they are called as backslash constants.

character	name	Meaning
<code>\a</code>	Bell	Beep sound
<code>\b</code>	Backspace	Cursor moves towards left by one position
<code>\n</code>	Newline	Cursor moves to next line
<code>\t</code>	Horizontal tab	Cursor moves towards right by 8 position
<code>\r</code>	Carriage return	Cursor moves towards beginning of the same line
<code>\"</code>	Double quotes	Double quotes
<code>\'</code>	Single quotes	Single quotes
<code>\?</code>	Question mark	Question mark
<code>\\</code>	Backslash	Backslash

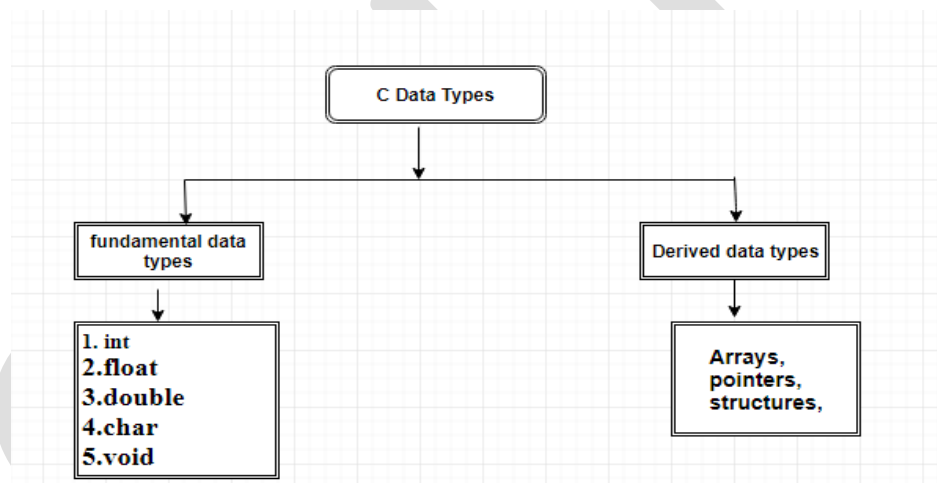
\0	NULL	
----	------	--

5.String constant

- A sequence of characters enclosed within pair of double quotes is called string constant.
- The string always ends with a NULL character.
- Ex: “9”
- “SVIT”

1.4.4 Data types and sizes

- **Definition:** The data type defines the type of data stored in memory location or the type of data the variable can hold.
- The classification of data types are:



- There are few basic data types supported in C.

1. **int**
2. **float**
3. **double**
4. **char**
5. **void**

1.Integer data type (int):

- It stores an integer value or integer constant.
- An int is a keyword using which the programmer can inform the compiler that the data associated with this keyword should be treated as integer.
- C supports 3 different types of integer:
 - short int,
 - int,

- long int

n-bit machine	Type	size	Range of unsigned int 0 to 2^n-1	Range of signed int be -2^n to $+2^n -1$
16-bit machine	short int	2 bytes	0 to $2^{16}-1$ 0 to 65535	-2^{15} to $+2^{15} -1$ -32768 to +32767
	int	2 bytes	0 to $2^{16}-1$ 0 to 65535	-2^{15} to $+2^{15} -1$ -32768 to +32767
	long int	4 bytes	0 to $2^{32}-1$ 0 to 4294967295	-2^{31} to $+2^{31} -1$ -2147483648 to + 2147483647

2. Floating data type (float):

- An float is a keyword using which the programmer can inform the compiler that the data associated with this keyword should be treated as floating point number.
- The size is 4 bytes.
- The range is **-3.4e38 to +3.4e38**.
- Float can hold the real constant accuracy up to 6 digits after the decimal point.

3.Double data type (double):

- An double is a keyword using which the programmer can inform the compiler that the data associated with this keyword should be treated as long floating point number.
- The size is 8 bytes.
- The range is from **1.7e-308 to 1.7 e+308**.
- Double can hold real constant up to 16 digits after the decimal point.

4.Character data type (char):

- char is a keyword which is used to define single character or a sequence of character in c language capable of holding one character from the local character set.
- The size of the character variable is 1 byte.
- The range is from **-128 to +127**.
- Each character stored in the memory is associated with a unique value termed as ASCII (American Standard Code for Information Interchange).
- It is used to represent character and strings.

n-bit machine	Type	size	Range of unsigned char	Range of signed char
			0 to $2^n - 1$	be -2^n to $+2^n - 1$
16-bit machine	short int	2 bytes	0 to $2^8 - 1$ 0 to 255	-2^7 to $+2^7 - 1$ -128 to +127

5. void data type (void):

- It is an empty data type.
- No memory is allocated.
- Mainly used when there are no return values.

1.4.4 variable

- A variable is a name given to memory location where data can be stored.
- Using variable name data can be stored in a memory location and can be accessed or manipulated very easily.

• Rules for variables

- The First character should be an **alphabet** or an **underscore** _
- Then First character is followed by any number of **letters or digits**.
- No extra symbols are allowed other than **letters ,digits and Underscore**
- Keywords cannot be used as an identifier

Example:

Sum – valid

For1- valid

for -invalid (it is a keyword)

1.4.4.1 Declaration of variables:

- Giving a name to memory location is called **declaring a variable**.
- Reserving the required memory space to store the data is called **defining a variable**.
- General Syntax:

datatype variable; (or)

datatype variable1, variable2,.....variablen;

example: **int a;**

float x, y;

double sum;

- From the examples we will come to know that “a” is a variable of type integer and allocates 2 bytes of memory. “x” and “y” are two variable of type float which will be allocated 4 bytes of memory for each variable. “sum” is a double type variables which will be allocated with 8 bytes of memory for each.

1.5 Variable Initialization

- Variables are not initialized when they are declared and defined, they contain garbage values(meaningless values)
- The method of giving initial values for variables before they are processed is called variable initialization.
- General Syntax:
Var_name = expr;
Where,
Var_name is the name of the variable ,
expr is the value of the expression.
- The “expr” on the right hand side is evaluated and stored in the variable name (Var_name) on left hand side.
- The expression on the right hand side may be a constant, variable or a larger formula built from simple expressions by arithmetic operators.

Examples:

- `int a=10;`
`// assigns the value 10 to the integer variable a`
- `float x;`
`x=20;`
`// creates a variable y of float type and assigns value 20 to it.`
- `int a=10,b,b=a;`
`// creates two variables a and b. “a” is assigned with value 10, the value of “a” is assigned to variable “b”. Now the value of b will be 10.`
- `price = cost*3;`
`//assigns the product of cost and 3 to price.`
- `Square = num*num;`
`// assigns the product of num with num to square.`

1.6 (OUTPUT FUNCTION)

Displaying Output using printf

- printf is an output statement in C used to display the content on the screen.
- print: Print the data stored in the specified memory location or variable.
- Format: The data present in memory location is formatted in to appropriate data type.
- There are various forms of printf statements.

Method 1: **printf(" format string");**

- Format string may be any character. The characters included within the double quotes will be displayed on the output screen
- Example: **printf("Welcome to India");**

Output:

Welcome to India

Method 2:

printf(" format string", variable list);

- Format string also called as control string.
- Format string consist of **format specifier** of particular data type
- Format specifiers starts with % sign followed by **conversion code**.
- variable list are the **variable names** separated by **comma**.
- Example:

```
int a=10;
```

```
float b=20;
```

```
printf(" integer =%d, floating=%f",a,b);
```

○ output:

integer=10, floating=20.00000

- Number of format specifiers must be equal to number of variables.
- While specifying the variables name make sure that it matches to the **format specifiers** with in the double quotes.

Format Specifiers

- Format specifiers are the character string with **% sign** followed with a character.
- It specifies the type of data that is being processed.
- It is also called **conversion specifier or conversion code**.
- There are many format specifiers defined in C.

Symbols	Meaning

%d	Decimal signed integer number
%f	float point number
%c	Character
%o	octal number
%x	hexadecimal integer(Lower case letter x)
%X	hexadecimal integer(Upper case letter X)
%e	floating point value with exponent(Lower case letter e)
%E	floating point value with exponent (Upper case letter E)
%ld	long integer
%s	String
%lf	double

1.7 Input Function (scanf)

Inputting Values Using scanf

- To enter the input through the input devices like keyboard we make use of scanf statement.

General Syntax:

scanf(“format string”, list of address of variables);

- Where: Format string consists of the access specifiers/format specifiers.
- Format string also called as control string.
- Format string consist of **format specifier** of particular data type
- Format specifiers starts with **%** sign followed by **conversion code**.
- **List of addresses of variables consist of the variable name preceded with & symbol(address operator).**

Example: int a;
 float b;
 scanf(“%d%f”,&a,&b);

Rules for scanf

- **No escape sequences or additional blank spaces** should be specified in the format specifiers.
- Ex: `scanf(“%d %f”,&a,&b); //invalid`
`scanf(“%d\n%f”,&a,&b); //invalid`
- **& symbol is must to read the values**, if not the entered value will not be stored in the variable specified. Ex: `scanf(“%d%f”,a,b);//invalid`.

A Simple C Program

Example 1:Let us discuss a simple program to print the Hello SVIT

```
/*program to print Hello svit*/
#include<stdio.h>
void main()
{
    printf(“ Hello SVIT”);
}
```

Output: Hello SVIT

Example 2:Consider another example program to find the simple interest

```
#include<stdio.h>
void main()
{
    float p,t,r;
    float si;
    printf(“enter the values of p,t,r\n”);
    scanf(“%f%f%f”,&p,&t,&r);
    si = (p*t*r)/100;
    printf(“Simple Interest=%f”,si);
}
```

The above program illustrates the calculation of simple interest.

- Firstly we have declared the header files `stdio.h` for including standard input and output which will be `printf` and `scanf` statements.
- Next we start with the main program indicated by `void main()`

- Within the main program we declare the required variables for calculating the simple interest. So we declare p, t and r as float type and si as float type.
- After which the values to p, t and r are read from keyboard using scanf.
- Computed the simple interest by the formula $(p*t*r)/100$ and the result is assigned to si.
- Display the result si.

1.8 Structure of C Program

Comments/Documentation Section
Preprocessor Directives
Global declaration section
<pre>void main() [Program Header] { Local Declaration part Execution part Statement 1 ----- ----- Statement n }</pre>
User defined Functions

- **Comments/Documentation Section**
 - Comments are short explaining the purpose of the program or a statement.
 - They are non executable statements which are ignored by the compiler.

- The comment begins **with /* and ends with */**.
- The symbols /* and */ are called **comment line delimiters**.
- We can put any message we want in the comments.

Example: /* Program to compute Quadratic Equation */

Note: Comments are ignored by C compiler, i.e. everything within comment delimiters

- **Preprocessor Directives**

- Preprocessor Directives begins with a **#** symbol
- Provides instructions to the compiler to include some of the files before compilation starts.
- Some examples of header files are

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <math.h>
#include <stdlib.h>     Etc...
```

- **Global Declaration Section**

- There will be some variable which has to be used in anywhere in program and used in more than one function which will be declared as global.

- **The Program Header**

- Every program must contain a main() function.
- The execution always starts from main function.
- Every C program must have only one main function.

- **Body of the program**

- Series of statements that performs specific task will be enclosed within braces { and }
- It is present immediately after program header.

It consists of two parts

1. **Local Declaration part:** Describes variables of program

Ex: int sum = 0;

int a , b;

float b;

2. **Executable part:** Task done using statements.

All statements should end with semicolon(;

Subroutine Section: All user defined functions that are called in main function should be defined.

1.9 Algorithm

- It is a step by step procedure to solve a problem
- A sequential solution of any problem that is written in natural language.
- Using the algorithm, the programmer then writes the actual program.
- The main use of algorithm is to help us to translate English into C.
- It is an outline or basic structure or logic of the problem.

1.9.1 Characteristics of Algorithm

- Each and every instruction must be precise and unambiguous
- Each instruction execution time should be finite.
- There should be a termination condition.
- It has to accept 0 or more inputs and produce compulsory an output.
- It can accept any type of input and produce a corresponding output.

1.9.2 General Way of Writing the Algorithm

- Name of the algorithm must be specified.
- Beginning of algorithm must be specified as Start
- Input and output description may be included
- Step number has to be included for identification
- Each step may have explanatory note provided with in square bracket followed by operation.
- Completion of algorithm must be specified as end or Stop.

Ex 1: Write an algorithm to add two numbers

Algorithm: Addition of two numbers

Input: Two number a and b

Output: Sum of two numbers

Step 1: Start

Step 2: [input two number]

 Read a and b

Step 3: [compute its addition]

Sum = a + b

Step 4: print Sum

Step 5: Stop

Ex 2: Write an algorithm to compute simple interest

Algorithm: To find Simple Interest

Input: p, t, r

Output: SI

Step 1: Start

Step 2: [input a principle p, time t and rate r]

Read p, t, r

Step 3: [compute simple interest]

$SI \leftarrow (p*t*r)/100$

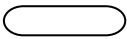
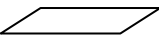
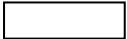

Step 4: display SI


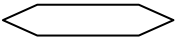
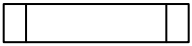


Step 5: Stop

1.10 Flow charts

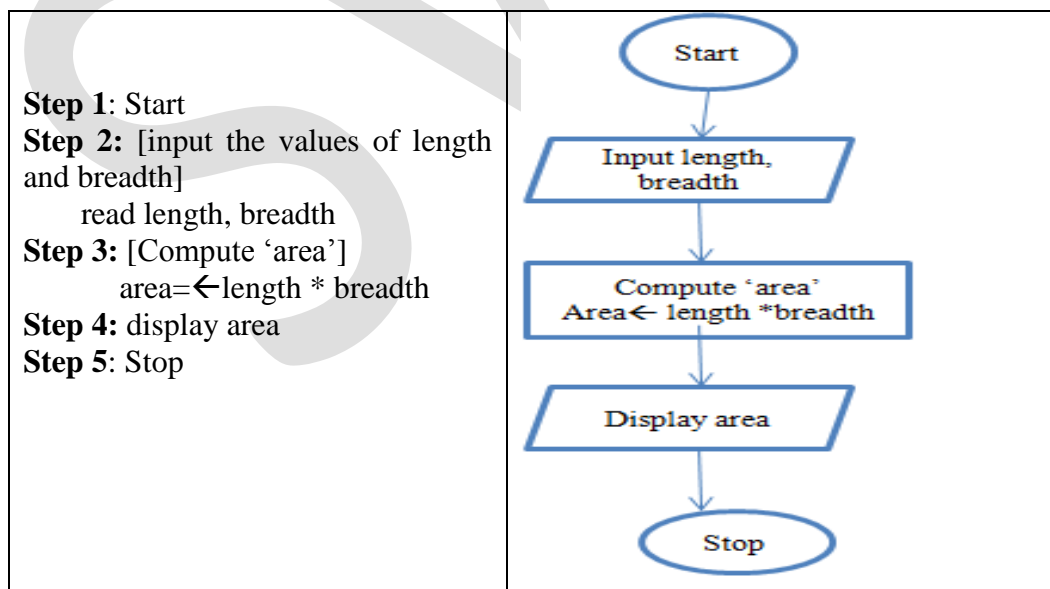
- A flowchart is a pictorial or graphical representation of an algorithm or a program.
- It consist of sequences of instructions that are carried out in an algorithm.
- The shapes represent operations.
- Arrows represent the sequence in which these operations are carried out.
- It is mainly used to help the programmer to understand the logic of the program.

1.10.1 Notations used in flowchart

Shape	Name	Meaning
	Start and Stop	Used to denote start and stop of flowchart
	Input and Output	Parallelogram used to read input and display output
	Processing/computation	Rectangle used to include instructions or processing statements
	Decision/ Condition	To include decision making statements

		rhombus is used
	Connector	Join different flow chart or flow lines
	Repetition/ Loop	Hexagon is used for looping constructs
	Functions	To include functions in a program, the name of function should be enclosed with in the figure
	Control Flow Lines	Arrows denoting flow of control in flow chart
	Comment box	For including the comments, definition or explanation

Ex 1. Algorithm and Flowchart to Input the dimensions of a rectangle and print its area



1.11 Operators and Expressions

1.11.1 Operators:

- “An operator is a symbol that specifies the operation to be performed on various types of operands”. Or in other words “An operator is a symbol that tells the computer to perform certain mathematical or logical manipulations”.
- Operators are used in programs to manipulate data and variable.

Example: +, -, *

1.11.2 Operand:

- The entity on which a operator operates is called an operand. Here the entity may be a constant or a variable or a function.
- An operator may have one or two or three operands.

Example: a+b-c*d/e%f

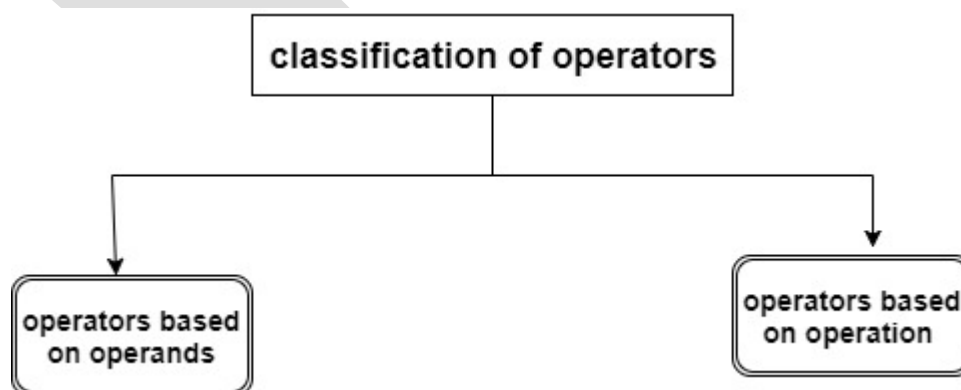
In this example there are **6 operands** ie a, b, c, d, e, f and **5 operators** i.e +, -, *, / and %.

1.11.3 Expression:

- A combination of operands and operators that reduces to a single value is called an expression.

Eg: a+b/d

Classification of operators



1.12.1 Classification of operators based on operands

- The Operators are classified into **four** major categories based on the number of operands as shown below:

- 1) *Unary Operators*
- 2) *Binary Operators*
- 3) *Ternary Operators*
- 4) *Special Operators*

1. **Unary Operators:** An operator which acts on only **one operand** to produce the result is called unary operator. The Operators precede the operand.

Examples: **-10**
 -a
 --b

2. **Binary Operators:** An operator which acts on **two operands** to produce the result is called a binary operator. In an expression involving a binary operator, the operator is in between two operands.

Examples: a + b a*b

3. **Ternary Operator:** An operator which acts on **three operands** to produce a result is called a ternary operator.

- Ternary operator is also called **conditional operator**.
- **Example:** **a ?b : c**
- Since there are three operands a, b and c (hence the name ternary) that are associated with operators ?and : it is called ternary operator.

1.12.2 C operators can be classified based on type of operation

1. **Arithmetic operators**
2. **Relational operators**
3. **Logical operators**
4. **Assignment operators**
5. **Increment and decrement operators**
6. **Conditional operators**
7. **Bitwise operators**
8. **Special operators**

1) **Arithmetic operators**

- C provides all basic arithmetic operators.

- The operators that are used to perform arithmetic operation such as addition, subtraction, multiplication, division etc are called arithmetic operators.
- The operators are +, -, *, /, %.
- Let a=10, b=5. Here, **a and b are variables and are known as operands.**

Description	Symbol	Example	Result	Priority	Associativity
Addition	+	a+b = 10 + 5	15	2	L-R
Subtraction	-	a-b = 10 - 5	5	2	L-R
Multiplication	*	a* b = 10 * 5	50	1	L-R
Division(Quotient)	/	a/b = 10/5	2	1	L-R
Modulus(Remainder)	%	a % b = 10 % 5	0	1	L-R

/*Program to illustrate the use of arithmetic operators is given below*/

```
#include<stdio.h>
void main()
{
    int a,b,sum,diff,mul,divres,modres;
    printf("enter a and b value\n");
    scanf("%d %d",&a,&b);
    sum=a+b;
    diff=a-b;
    mul=a*b;
    divres=a/b;
    modres=a%b;
    printf("%d, %d, %d, %d, %d\n", sum,diff,mul,divres,modres);
}
```

Output: 35,25,150,6,0

Converting Mathematical Expression into C Expression

Mathematical Expression	C expression
-------------------------	--------------

$a \div b$	a/b
$S = \frac{a+b+c}{2}$	$S = (a+b+c)/2$
$Area = \sqrt{s(s-a)(s-b)(s-c)}$	$Area = \text{sqrt}(s*(s-a)*(s-b)*(s-c))$
ax^2+bx+c	$a*x*x+b*x+c$
$e^{ a }$	$\text{exp}(\text{abs}(a))$

2) Relational Operators

- We often compare two quantities depending on their relation, take certain decisions.
- For example, we compare the age of two persons, or the price of two items, and so on. These comparisons can be done with the help of relational operators.
- The operators that are used to find the relationship between two operands are called as **relational expression**.
- An expression such as **$a < b$ or $1 < 20$**
- **The value of relational expression is either one or zero.**
- It is **one** if the specified relation is **true** and zero if the relation is **false**.

Example: $10 < 20$ is true.

$20 < 10$ is false.

Relational operator in C are:

Operator	Meaning	Precedence	Associativity	Example	Result
<	is less than	1	L->R	$20 < 10$	F
<=	is less than or equal to	1	L->R	$20 <= 10$	F
>	is greater than	1	L->R	$20 > 10$	T
>=	is greater than or equal to	1	L->R	$20 >= 10$	T
==	is Equal to	2	L->R	$20 == 10$	F
!=	is not equal to	2	L->R	$20 != 10$	T

Example:

```

a=10 b=20 c=30
a > b != c <= b > a
10 > 20 != 30 <= 20 > 10
0 != 30 <= 20 > 10
0 != 0 > 0
0 != 0
    
```

3) Logical Operators

- The operator that are used to combine two or more relational expression is called logical operators.
- Result of logical operators is always true or false.

Logical Operators in C are.

Operator	Meaning	Precedence	Associativity
&&	logical AND	2	L->R
	logical OR	3	L->R
!	logical NOT	1	L->R

3.1 logical AND (&&)

The output of logical operation is true if and only if both the operands are evaluated to true.

operand 1	AND	operand 2	result
True (1)	&&	True(1)	True(1)
True (1)	&&	False(0)	False(0)
False(0)	&&	True(1)	False(0)
False(0)	&&	False(0)	False(0)

Example:

36 && 0

=1 && 0

=0(False)

3.2 logical OR (||)

The output of logical operation is false if and only if both the operands are evaluated to false.

operand 1	OR	operand 2	result
True (1)		True(1)	True(1)
True (1)		False(0)	True(1)
False(0)		True(1)	True(1)
False(0)		False(0)	False(0)

Example:

36 || 0
 =1 || 0
 =0(True)

3.4 logical NOT

It is a unary operator. The result is true if the operand is false and the result is false if the operand is true.

operand	! operand
True (1)	False(0)
False(0)	True (1)

Example:

(a) !0=1

(b) !36

!1=0

NOTE: The logical operators && and || are used when we want to test more than one condition and make decisions.

An example: a>b && x==10

The above logical expression is true only if a>b is true and x==10 is true. If either (or both) of them are false, the expression is false.

4) Assignment Operators

➤ An operator which is used to assign the data or result of an expression into a variable is called Assignment operators. The usual assignment operator is =.

➤ Types of Assignment Statements:

- 1.Simple assignment statement
- 2.Short Hand assignment statement
- 3.Multiple assignment statement

1.Simple assignment statement

Syntax: **Variable=expression;**

Ex:

a= 10; //RHS is constant

a=b; // RHS is variable.

a=b+c; //RHS is an expression

So,

- 1.A expression can be a constant,variable or expression.
- 2.The symbol = is assignment operator
- 3.The expression on right side of assignment is evaluated and the result is converted into type of variable on left hand side of Assignment operator.
- 4.The converted data is copied into variable which is present on left hand side(LHS) of assignment operator.

Example:

```
int a;  
a=100;        // the value 100 is assigned to variable a
```

```
int a=100;  
int b;  
b=a;        // a value i.e 100 is assigned to b
```

2.Short Hand assignment Statement

C has a set of “**shorthand**” assignment operators of the form.

var op = exp;

where

v is a variable, exp is an expression and op is a C library arithmetic operator.

The operator **op=** is known as the shorthand assignment operator.

The assignment statement

var op= exp; is equivalent to $v = \text{var op} (\text{exp})$. with var evaluated only once.

The operators such as +=, -=, *=, /= are called assignment operators.

Example:

The statement

$x = x + 10;$ can be written as $x += 10;$

Example:

Solve the equation $x^* = y + z$ when $x=10, y=5$ and $z=3$

Solution: $x^* = y + z$ can be written as

$$x = x^*(y + z)$$

$$x = 10^*(5 + 3)$$

$$x = 10^*8$$

$$x = 80$$

shorthand assignment	Meaning	Description
a+=2	a=a+2	Find a+2 and store result in a
a-=2	a=a-2	Find a-2 and store result in a
a*=2	a=a*2	Find a*2 and store result in a
a/=2	a=a/2	Find a/2 and store result in a

3. Multiple Assignment Statement

A statement in which a value or set of values are assigned to more than one variable is called multiple assignment statement.

Example:

$i=10;$

```
j=10;
```

```
k=10;
```

can be written as:

```
i=j=k=10; //Multiple assignment statement
```

Note: Assignment operator is right associative operator. Hence in the above statement 10 is assigned to k first, the k is assigned to j and j is assigned to i.

5) Increment and Decrement Operators

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

++ and --

- The operator ++ adds 1 to the operand, while -- subtracts 1. The increment and decrement operators are Unary operators.
- The increment and decrement operators are classified into two categories:

1. Post Increment and Post Decrement:

- If increment or decrement operator is placed immediately after the operand is called Post Increment and Post Decrement.
- As the name indicates Post Increment and Post Decrement means increment or decrement the operand value is used.
- So operand value is used first and then the operand value is incremented or decremented by 1.

Example:

Post Increment: a++ equivalent to a=a+1

Post Decrement: a-- equivalent to a=a-1

Post Increment:

Eg 1:

```
#include<stdio.h>

void main()
{
    int a=20;
    a++;
    printf("%d",a);
}
```

Output:21

Description: Initial value of a is 20 after execution of a++, the value of a will be 21.

Eg2:

```
#include<stdio.h>
void main( )
{
int a=20,b;
b=a++;
printf("a=%d",a);
printf("b=%d",b);
}
```

Output : a= 21

b=20

Post Decrement:

Eg 1:

```
#include<stdio.h>
void main()
{
int a=20;
a--;
printf("%d",a);
}
```

Output:19

Description: Initial value of a is 20 after execution of a--, the value of a will be 19.

2.Pre Increment and Pre Decrement

- If increment or decrement operator is placed before the operand is called Pre Increment and Pre Decrement.
- As the name indicates Pre Increment and Pre Decrement means increment or decrement before the operand value is used.
- So the operand value is incremented or decremented by 1 first then operand value is used .

Example:

Post Increment: ++a equivalent to a+1=a

Post Decrement: a--equivalent to a-1=a

Pre Increment:

Eg 1:

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

Output:21

```

void main()
{
    int a=20;
    ++a;
    printf("%d",a);
}

```

Description: Initial value of a is 20 after execution of ++a, the value of a will be 21.

Pre Decrement:

Eg 1:

```

#include<stdio.h>
void main()
{
    int a=20;
    --a;
    printf("%d",a);
}

```

Output:19

Description: Initial value of a is 20 after execution of --a, the value of a will be 19.

Consider the following(let a=20)

Post increment

b=a++

current value of a used, **b=a //b=20**
a is incremented by 1 **a=a+1 //a=21**

Pre increment

b=++a

a is incremented by 1, **a=a+1 //a=21**
incremented value of a used, **b=a //b=21**

Post Decrement (let a=10)

b=a--

current value of a used, **b=a //b=10**
a is decremented by 1 **a=a-1 //a=9**

Pre decrement(let a=10)

b=--a

a is decremented by 1, **a=a-1 // a=9**
decremented value of a used, **b=a // b=9**

6) Conditional Operator (OR) Ternary Operator

The operator which operates on 3 operands are called Ternary operator or conditional operator.

Syntax: (exp1)? exp2: exp3

where exp1,exp2, and exp3 are expressions and

- exp1 is an expression evaluated to true or false
- if exp1 is evaluated to true exp2 is executed
- if exp1 is evaluated false exp3 is executed

For example, consider the following statements.

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

In this example, Max will be assigned the value of b i.e b=15. This can be achieved using the if... else statements as follows:

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

7) Bitwise Operators

Bitwise operators are used to manipulate the bits of given data. These operators are used for testing the bits, or shifting them right or left. Bitwise operators may not be applied to float or double.

Operator	Meaning	Precedence	Associativity
~	Bitwise Negate	1	R->L
<<	Bitwise Left Shift	2	L->R
>>	Bitwise Right Shift	2	L->R
&	Bitwise AND	3	L->R
^	Bitwise XOR(exclusive OR)	4	L->R
	Bitwise OR	5	L->R

1)Bitwise AND (&)

If the corresponding bit positions in both the operand are 1,then AND operation results in 1 ;otherwise 0.

operand 1	AND	operand 2	result
0	&	0	0
0	&	1	0
1	&	0	0
1	&	1	1

a=10 00001010

b=6 00000110 (a&b)

c=2 00000010

2)Bitwise OR(|)

If the corresponding bit positions in both the operand are 0,then OR operation results in 0 ;otherwise 1.

operand 1	OR	operand 2	result
0		0	0
0		1	1
1		0	1
1		1	1

a=10 00001010

b=6 00000110 (a|b)

c=2 00001110

3)Bitwise XOR(^)

If the corresponding bit positions in both the operand are different,then XOR operation results in 1 ;otherwise the result is 0.

operand 1	XOR	operand 2	result
0	^	0	0
0	^	1	1
1	^	0	1
1	^	1	0

```

a=10  00001010
b=6   00000110 (a^b)
-----
c=12  00001100

```

4)Left shift operator(<<)

The operator that is used to shift the data by a specified number of bit positions towards left is called left shift operator.

After left shift MSB should be discarded and LSB will be empty and should be filled with "0".

```

b=a<<num  —————> Second operand
  |
  |
  v
  First operand

```

The first operand is the value to be shifted.

The second operand specifies the number of bits to be shifted.

The content of first operand are shifted towards left by the number bit positions specified in second operand.

Example:

```

#include<stdio.h>
void main()

```

OUTPUT: 5<<1=10


```

int a=10,b;
b=~a;
printf("Negation of a is %d",b);
}

```

Tracing:

```

a=10      0 0 0 0 1 0 1 0
b=245    1 1 1 1 0 1 0 1

```

8) Special Operators

C supports some special operators of interest such as comma operator, sizeof operator.

The Comma Operator

- ❖ The comma operator can be used to link the related expressions together. A comma-linked list of expressions are evaluated from left to right. The comma operator has least precedence among all operator.
- ❖ It is a binary operator which is used to:

1. Separate items in the list.

Eg: a=12,35,678;

2. Combine two or more statements into a single statement.

Eg: int a=10;
 int b=20;
 int c=30;

can be combined using comma operator as follows:

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

Example, the statement: value = (x=10,y=5,x+y)

first assigns the value 10 to x, then assigns 5 to y, and finally assigns 15 to value. Since comma operator has the lowest precedence of all operators, the parentheses are necessary.

The sizeof() Operator

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

Syntax: **sizeof(operand)**

Examples:

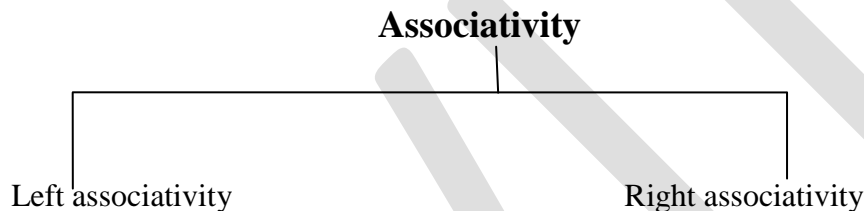
- | | |
|------------------|---------------|
| 1. sizeof(char); | Output:1 Byte |
| 2. int a; | Output:2 Byte |
| sizeof(a) | |

Precedence[Priority]

It is the rule that specifies the order in which certain operations need to be performed in an expression. For a given expression containing more than two operators, it determines which operations should be calculated first.

Associativity

If all the operators in an expression have equal priority then the direction or order chosen left to right or right to left to evaluate an expression is called associativity.



Left associativity:

In an expression if two or more operators having same priority are evaluated from left to right then the operators are called left to right associative operator, denoted as **L->R**

Right associativity:

In an expression if two or more operators having same priority are evaluated from left to right then the operators are called right to left associative operator, denoted as **R->L**

BODMAS Rule can be used to solve the expressions, where

B: Brackets O: Operators D: Division M: Multiplication A: Addition S: Subtraction

The Precedence (Hierarchy) of Operator

Operator Category	Operators in Order of Precedence (Highest to Lowest)	Associativity
Innermost brackets/Array elements	() , [] , {}	Left to Right(L->R)

reference		
Unary Operators	++, --, sizeof(), ~, +, -	Right to Left(R->L)
Member Access	-> or *	L->R
Arithmetic Operators	*, /, %	L->R
Arithmetic Operators	-, +	L->R
Shift Operators	<<, >>	L->R
Relational Operators	<, <=, >, >=	L->R
Equality Operators	==, !=	L->R
Bitwise AND	&	L->R
Bitwise XOR	^	L->R
Bitwise OR		L->R
Logical AND	&&	L->R
Logical OR		L->R
Conditional Operator	?:	R->L
Assignment Operator	=, +=, -=, *=, /=, %=	R->L
Comma Operator	,	L->R

Modes of Arithmetic Expressions

- An expression is a sequence of operands and operators that reduces to a single value. Expressions can be simpler or complex. An operator is a syntactical token that requires an action to be taken. An operand is an object on which an operation is performed.
- Arithmetic expressions are divided into 3 categories:
 1. Integer Expressions
 2. Floating Point Expressions
 3. Mixed mode Expressions

1. Integer Expressions

- If all the operands in an expression are integers then the expression is called Integer Expression.
- The result of integer expression is always integer.

Ex: $4/2=2$

2. Floating Point Expressions

- If all the operands in an expression are float numbers then the expression is called floating point Expression.
- The result of floating point expression is always float

Ex: $4.0/3.0=1.3333$

3. Mixed Mode Expressions

- An expression that has operands of different types is called Mixed Mode Expression.

Ex: $4.0/2$ i.e float/int

Evaluation of Expressions involving all the operators

The rules to be followed while evaluating any expressions are shown below.

- Replace the variables if any by their values.
- Evaluate the expressions inside the parentheses
- Rewrite the expressions with increment or decrement operator as shown below:
 - a) Place the pre-increment or pre-decrement expressions before the expression being evaluated and replace each of them with corresponding variable.
 - b) Place the post-increment or post-decrement expressions after the expression being evaluated and replace each of them with corresponding values.
- Evaluate each of the expression based on precedence and associativity.

1.10 Type Conversion

The process of converting the data from one data type to another data type is called Type conversion.

Type conversion is of two types

(i) Implicit type conversion

(ii) Explicit type conversion

i). Implicit Conversion

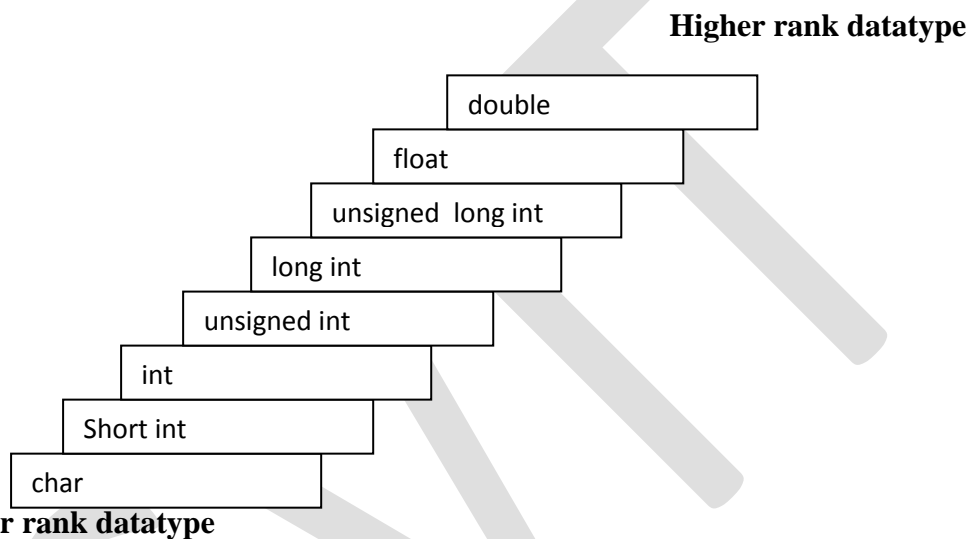
- C can evaluate the expression if and only if the data types of two operands are same.
- If operands are of different type, it is not possible to evaluate expression.
- In such situation to ensure that both operands are of same type, C compiler converts data from lower rank to higher rank automatically.

- This process of converting data from lower rank to higher rank is called as Implicit type conversion(ITC).

For example,

- If one operand type is same as other operand type, no conversion takes place and type of result remains same as the operands
i.e $\text{int} + \text{int} = \text{int}$, $\text{float} + \text{float} = \text{float}$ etc.
- If one operand type is int and other operand type is float, the operand with type int is promoted to float.

Promotion of hierarchy is as follows:



Examples:

- i. $5 + 3 = 8$
 $\text{int} + \text{int} = \text{int}$
- ii. $5 + 3.5 = 8.5$
 $\text{int} + \text{float} = \text{float}$

ii) Explicit type conversion

- If the operands are of same datatype no conversion takes place by compiler. Sometimes the type conversion is required to get desired results.
- In such situation programmer himself has to convert the data from one type to another. This forcible conversion from one datatype to another is called as explicit type conversion(ETC).

(type) expression

Syntax:**Example:** (int) 9.93=9

The data conversion from higher data type to lower data type is possible using explicit type conversion. The following example gives the explicit type conversion.

x=(int) 7.5	7.5 is converted to integer by truncation
a=(int)21.3 / (int) 4.5	Evaluated as 21/4 and the result would be 5.
b=(double) sum/n	Division is done in floating point mode
y=(int)(a+b)	The result a + b is converted to integer
z=(int)a +b	a is converted to integer and then added to b

“What is type casting? Explain with example?”

Definition: C allows programmers to perform typecasting by placing the type name in parentheses and placing this in front of the value.

For instance

```
main()
{
    float a;
    a = (float)5 / 3;
}
```

Gives result as 1.666666 . This is because the integer 5 is converted to floating point value before division and the operation between float and integer results in float.

Difference between ITC and ETC

ITC	ETC
<p>1.It is performed by Compiler.</p> <p>2.ITC performs only lower to higher data conversion</p> <p>3.ITC is performs when both the operands are of different type.</p> <p>4.Example 2+3.4=5.4</p>	<p>1.It is performed by programmer.</p> <p>2.ETC perform both lower to higher and higher to lower.</p> <p>3.ETC is performed when both the operands are of same type.</p> <p>4.Example float a=10.6 (int)a; Ans:10</p>

Question Bank

Module 1				
Sl.No	Questions	Appeared in VTU qp month/year	CO mappin g	BT Level Attained
1.	Define pseudocode .Write a pseudocode to find the sum and average of three numbers.	June/July 2016 Dec/Jan 2015	CO1	BT1

2	Classify the following identifiers as valid/invalid. a) num2 b) \$num1 c) +add d) a_2	June/July 2016	CO1	BT4
3	Explain scanf and printf functions with example.	June/July 2016	CO1	BT1
4	List all the operators used in C give examples	June/July 2016	CO1	BT1
5	Write the output for the following code. <pre> Void main() { Int a=5,b=2,res1; Float f1=5.0,f2=2.0,res2; Res1=5/2.0+a/2+a/b; Res2=f1/2*f1-f2; Printf("res1=%d res2=%f",res1,res2); } ii) void main() { int i=5,j=6,m,n; m = ++ i + j ++; n= --i + j--; printf("m=%d n=%d",m,n); } </pre>	June/July 2016	CO1	BT1
6	Explain the structure of C program with an programming example.	Dec/Jan 2015, Dec 2011	CO1	BT1
7	Explain any five operators used in C language	Dec/Jan 2015, Jun/July- 2015	CO1	BT1
8	What are type conversions? Explain two type of type conversions with example	Dec/Jan 2015, June 2012	CO1	BT1
9	write a program in C to find area and perimeter of rectangle	Dec/Jan 2015.- jun/july 2016	CO1	BT1
10	Define i) variable ii) constant iii) Associativity iv) Precedence (8M) - Dec/Jan 2015. What are data types? Mention the different data types supported by C language, giving an example to each	Jun/July- 2015	CO1	BT1
11	Write a c program which takes as input p,t,r and computes simple interest	Jun/July- 2015.	CO1	BT1

12	What is an operator? List and explain various types of operators	Jun/July-2015	CO1	BT1
13	What is a token? What are the different types of tokens available in C language? Explain	Jun/July-2015	CO1	BT1
14	What is the value of x in the following code segments? Justify your answers: i) int a,b; float x; a=4; b=5; x=b/a; ii) int a,b; float x; a=4; b=5; x=(float)b/a;	Jun/July-2015	CO1	BT1
15	What are identifiers? Discuss the rules to be followed while naming identifiers? Give examples.(06 marks)	June/July 2013 Jun/July-2015	CO1	BT1
16	Explain format specifiers used in scanf() function to read int , float,char,double and long int data types.(06 marks)	June/July 2013	CO1	BT1
17	Write a c program to swap values of two integers without using a third variable and write flowchart for the same.(06 marks)	June/July 2013	CO1	BT1
18	Find the result of each of the following expression with i=4,j=2,k=6,a=2.(10 marks) i)k*=i+j ii)j=i/=k iii)i%=i/3 iv)m=i+(j=2+k)	June/July 2013	CO1	BT1
19	Explain the scope of local and global variables with examples(04 marks)	June/July 2013	CO1	BT1
20	Briefly explain how to create and run the program(04 marks) –	Jan 2013	CO1	BT1
21	Explain 5 types of data with its range and values(04 marks) –	Jan 2013	CO1	BT1
22	Explain scanf() and printf() functions with syntax. (06 marks)	Jan 2013	CO1	BT1
23	What do you mean by type conversion? Explain explicit type conversion with examples(04 marks)	Jan 2013	CO1	BT1

24	Explain the following operators with examples (09 marks) i)conditional ii)bitwise iii)sizeof	Jan 2013	CO1	BT1
25	Determine the value of each of the following logical expressions where a=5,b=10 and c=-6.(03 marks) i)a>b&&a>c ii)b>15&&c<0 a>0 iii)(a/2.0==0.0&&b/2.0!=0.0) c<0.0	Jan 2013	CO1	BT1
26	What are c tokens? Mention then? and explain any two tokens(08 marks)	Dec 2011	CO1	BT1
27	What is a datatype? explain the basic data types available in c (04 marks)	Dec 2011	CO1	BT1
28	What are variables how they are declared(04 marks)	Dec 2011	CO1	BT1
29	Write a program to find area of triangle given three sides(06 marks)	Dec 2011	CO1	BT1
30	With examples illustrate any four programming errors (04 marks)	Dec 2011	CO1	BT1
31	Write an algorithm and flowchart to calculate factorial of a number(6 marks)	June 2012	CO1	BT1
32	Explain precedence and associativity of operators in c with an example(08 marks) –June 2012	June 2012	CO1	BT1
33	List and explain coding constant(06 marks)	June/July 2011	CO1	BT1
34	If a=2,b=8,c=4,d=10 ,what is value of each of the following(04 marks) i)a+b/c*d-c/a ii)(b/a)%c iii)a++ + b-- + d++	June/July 2011	CO1	BT1
35	Write a c program to convert degrees into radians accepting the value from the user(06 marks)	Jan 2011	CO1	BT1
36	Explain the following operators with examples(08 marks) a. Relational ii)conditional iii)increment iv)special	May/June 2010	CO1	BT1

	operators			
37	What is the value of x if a=10,b=15 and (x>b)?a:b;(02 marks)	-Dec 2010	CO1	BT1
38	What would be the value of c if a=1,b=2?mention the steps involved(08 marks) C=(a>0&&a<=10)?a+b:a/b b. C=(a<0&&a<=10)?a+b:a/b	-Dec 2010	CO1	BT1
39	Draw a neat flowchart to exchange two numbers without using temporary variable(5 marks)	-Dec 2010	CO1	BT1
40	Evaluate the following expressions independent to each other, the declaration and initialization is as follows(05 marks): int i =3, j=4, k=2; i) i ++ -j-- ii) ++k %--j iii) j+1/i-1 iv) j++/i-- v) ++i/++j+1	-Dec 2010	CO1	BT5
41	What are escape sequences? why they are used? give examples(04 marks)	-June/July 08	CO1	BT1
42	What do you mean by mixed mode operation? Explain with an example(06 marks)	Dec 2010	CO1	BT1
43	Write a C program to find area of circle	Dec 2010	Co1	BT1
	Dec 2016/Jan 2017			
44	Define an Algorithm .write an algorithm to find area and perimeter of a rectangle.6M			
45	Write general structure of C ?Explain with an example. 6M			
46	Explain different types of input output function in c with syntax and examples. 6M			
47	Explain the following operators : Unary,binary conditional 6M			
48	Write an flowchart and C program to find simple interest. 4M			
	JUNE/JULY 2017			
49	Define pseudo code .Explain with an example. 5M			
50	Write a c program to find biggest among three numbers using ternery operator (5M)			

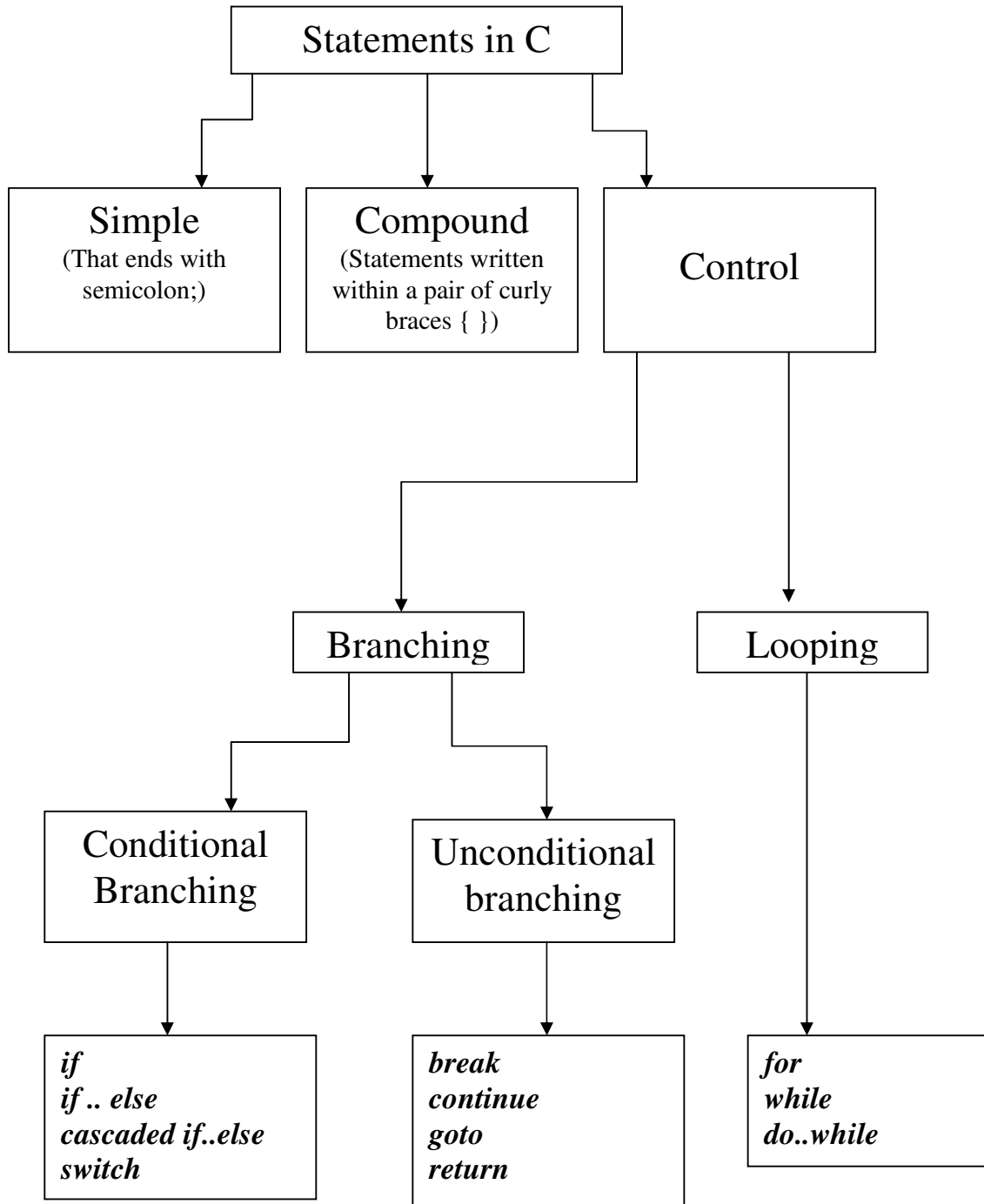
51	Explain the following constants with example(6M) i)integer constant ii)floating constant iii)character constant.			
52	List the formatted input/output functions of c language.explain the basic structure of c program with proper syntax and example.(6M)			
53	Define an algorithm. write an algorithm to find the area of circle and triangle.(6M)			
54	Evaluate the following expression: i) $22+3<6\&\&!5 22==7\&\&22-2>=5$ ii) $a+2>b !c \&\&a==d a-2<=e$ where a=11,b=6,c=0,d=7,and e=5			
55	List all the restrictions on the variable names(06 Marks)			
56	Explain the block structure of a "C Program" (08 Marks)			
57	What are the basic data types available in "C"? Write the significance of each datatype (06 Marks)			
58	What is an assignment statement? Give the general form of an assignment statement? (5 Marks)			
59	Explain with example, the various constants available in 'C' program(5 Marks)			
60	List and explain any five operators used in "C" programming Language(10 marks)			

MODULE II

2. BRANCHING AND LOOPING:

- 2.1 Two way selection (if, if-else, nested if-else, cascaded if-else)
- 2.2 switch statement
- 2.3 Ternary operator?
- 2.4 Loops (for, do-while, while) in C
- 2.5 break , continue and goto
- 2.6 Programming examples and exercises.

Statements in C



2.1 Two way selection (if, if-else, nested if-else, cascaded if-else)

THE if STATEMENT

This is basically a “one-way” decision statement.

This is used when we have only one alternative.

The syntax is shown below:

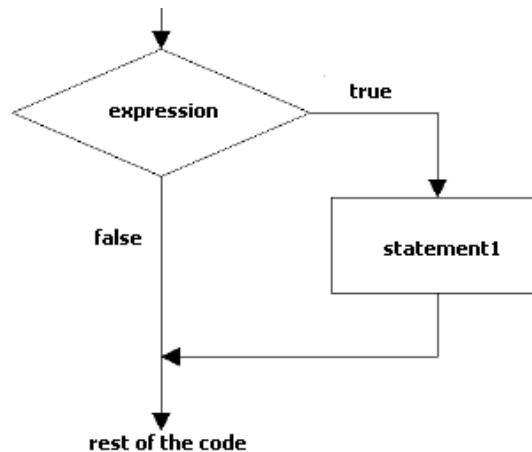
```
if(expression)
{
    statement1;
}
```

Firstly, the expression is evaluated to *true* or *false*.

If the expression is evaluated to *true*, then statement1 is executed.

If the expression is evaluated to *false*, then statement1 is skipped.

The flow diagram is shown below:



Example: Program to illustrate the use of if statement.

```
#include<stdio.h>
void main()
{
    int n;
    printf("Enter any non zero integer: \n") ;
    scanf("%d", &n)
    if(n>0)
        printf("Number is positive number ");
    if(n<0)
        printf("Number is negative number ");
}
```

Output:

Enter any non zero integer:

7

Number is positive number

THE if else STATEMENT

This is basically a “two-way” decision statement.

This is used when we must choose between two alternatives.

The syntax is shown below:

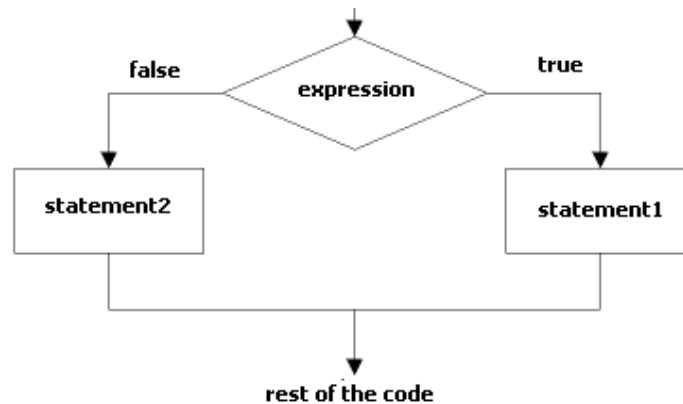
```
if(expression)
{
    statement1;
}
else
{
    statement2;
}
```

Firstly, the expression is evaluated to true or false.

If the expression is evaluated to *true*, then statement1 is executed.

If the expression is evaluated to *false*, then statement2 is executed.

The flow diagram is shown below:



Example: Program to illustrate the use of if else statement.

```
#include<stdio.h>
void main()
{
    int n;
    printf("Enter any non-zero integer: \n") ;
    scanf("%d", &n)
    if(n>0)
        printf("Number is positive number");
    else
        printf("Number is negative number");
}
```

Output:

Enter any non-zero integer:

7

Number is positive number

THE nested if STATEMENT

An if-else statement within another if-else statement is called nested if statement.

This is used when an action has to be performed based on many decisions. Hence, it is called as multi-way decision statement.

The syntax is shown below:

```

if(expr1)
{
    if(expr2)
        statement1
    else
        statement2
}
else
{
    if(expr3)
        statement3
    else
        statement4
}

```

- Here, firstly *expr1* is evaluated to true or false.

If the *expr1* is evaluated to *true*, then *expr2* is evaluated to true or false.

If the *expr2* is evaluated to *true*, then *statement1* is executed.

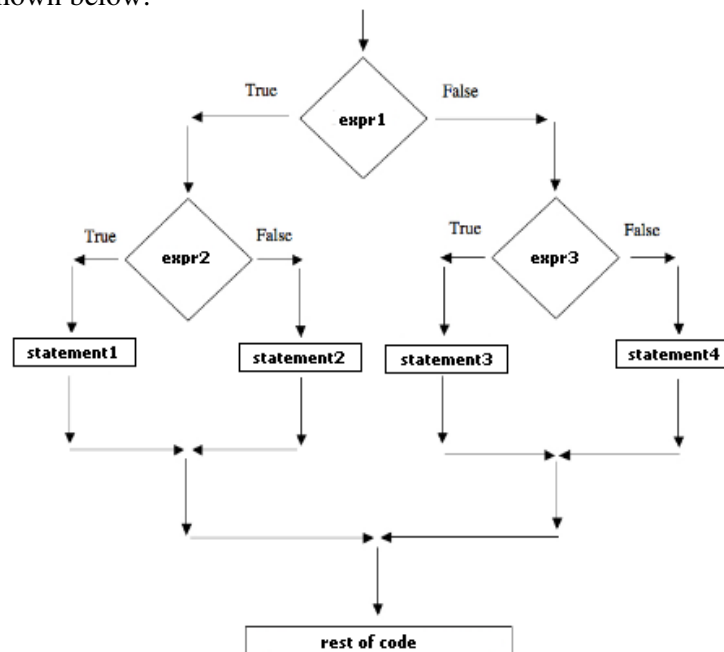
If the *expr2* is evaluated to *false*, then *statement2* is executed.

If the *expr1* is evaluated to *false*, then *expr3* is evaluated to true or false.

If the *expr3* is evaluated to *true*, then *statement3* is executed.

If the *expr3* is evaluated to *false*, then *statement4* is executed.

- The flow diagram is shown below:



Example: Program to select and print the largest of the 3 numbers using nested “if-else” statements.

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

Output:

```
Enter Three Values:
7 8 6
Largest Value is: 8
```

THE CASCADED if-else STATEMENT (THE else if LADDER STATEMENT)

This is basically a “multi-way” decision statement.

This is used when we must choose among many alternatives.

The syntax is shown below:

```
if(expression1)
{
    statement1;
}
else if(expression2)
{
    statement2;
}
else if(expression3)
{
    statement3
}
```

```
    else
    {
        default statement
    }
```

The expressions are evaluated in order (i.e. top to bottom).

If an expression is evaluated to true, then

→ Statement associated with the expression is executed &

→ Control comes out of the entire else if ladder

For ex, if expression1 is evaluated to *true*, then statement1 is executed.

If all the expressions are evaluated to false, the last statement4 (default case) is executed.

2.2 *switch* Statement

This is a multi-branch statement similar to the if - else ladder (with limitations) but clearer and easier to code.

```
Syntax :    switch ( expression )
            {
                case constant1 :    statement1 ;
                                    break ;

                case constant2 :    statement2 ;
                                    break ;

                ...

                default :    statement ;
            }
```

The value of expression is tested for equality against the values of each of the constants specified in the **case** statements in the order written until a match is found. The statements associated with that case statement are then executed until a break statement or the end of the switch statement is encountered.

When a break statement is encountered execution jumps to the statement immediately following the switch statement.

The default section is optional -- if it is not included the default is that nothing happens and execution simply falls through the end of the switch statement.

The switch statement however is limited by the following

- Can only test for equality with **integer constants** in case statements.
- No two case statement constants may be the same.
- Character constants are automatically converted to integer.

For Example :- Program to simulate a basic calculator.

```
#include <stdio.h>
void main()
{
    double num1, num2, result ;
    char op ;

    printf ( " Enter number operator number\n" ) ;
    scanf ("%f %c %f", &num1, &op, &num2 ) ;
    switch ( op )
    {
        case '+' : result = num1 + num2 ;
                    break ;
        case '-' : result = num1 - num2 ;
                    break ;
        case '*' : result = num1 * num2 ;
                    break ;
        case '/' : if ( num2 != 0.0 ) {
                        result = num1 / num2 ;
                        break ;
                    }
        // else we allow to fall through for error message

        default : printf ("ERROR -- Invalid operation or division
                        by 0.0" ) ;
    }
    printf( "%f %c %f = %f\n", num1, op, num2, result) ;
}
```

Note : The break statement need not be included at the end of the case statement body if it is logically correct for execution to fall through to the next case statement (as in the case of division by 0.0) or to the end of the switch statement (as in the case of default :).

2.3 Ternary operator- ? :

This is a special shorthand operator in C and replaces the following segment

```
if ( condition )
    expr_1 ;
else
    expr_2 ;
```

with the more elegant:

Syntax: *condition ? expr_1 : expr_2 ;*

The ?: operator is a ternary operator in that it requires three arguments. One of the advantages of the ?: operator is that it reduces simple conditions to one simple line of code which can be thrown unobtrusively into a larger section of code.

For Example :- to get the maximum of two integers, x and y, storing the larger in max.

```
max = x >= y ? x : y ;
```

The alternative to this could be as follows

```
if ( x >= y )
    max = x ;
else
    max = y ;
```

giving the same result but the former is a little bit more succinct.

2.4 Loops (*for*, *do-while*, *while*) in C

for statement

The *for* statement is most often used in situations where the programmer knows in advance how many times a particular set of statements are to be repeated. The *for* statement is sometimes termed a counted loop.

```
Syntax :    for ( [initialisation] ; [condition] ; [increment] )
              {
                [statement body] ;
              }
```

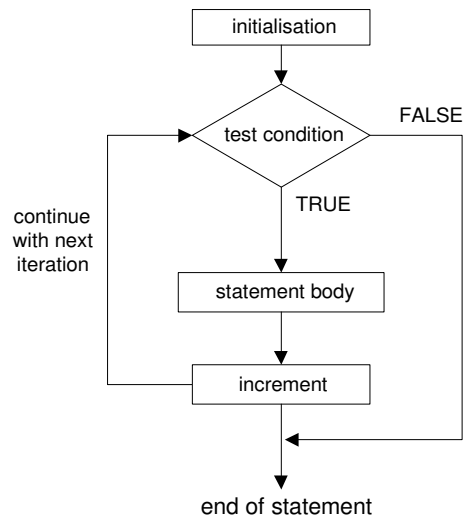
initialisation :- this is usually an assignment to set a loop counter variable for example.

condition :- determines when loop will terminate.

increment :- defines how the loop control variable will change each time the loop is executed.

statement body :- can be a single statement, no statement or a block of statements.

The *for* statement executes as follows :-



Note : The square braces above are to denote optional sections in the syntax but are not part of the syntax. The semi-colons must be present in the syntax.

For Example : Program to print out all numbers from 1 to 100.

```

#include <stdio.h>

void main()
{
    int x ;

    for ( x = 1; x <= 100; x++ )
        printf( "%d\n", x ) ;
}
  
```

Curly braces are used in C to denote code blocks whether in a function as in main() or as the body of a loop.

For Example :- To print out all numbers from 1 to 100 and calculate their sum.

```

#include <stdio.h>

void main()
{
    int x, sum = 0 ;

    for ( x = 1; x <= 100; x++ )
    {
        printf( "%d\n", x ) ;
        sum += x ;
    }
    printf( "\n\nSum is %d\n", sum ) ;
}
  
```


Multiple Initialisations

C has a special operator called the **comma operator** which allows separate expressions to be tied together into one statement.

For example it may be tidier to initialise two variables in a for loop as follows :-

```
for ( x = 0, sum = 0; x <= 100; x++ )
{
    printf( "%d\n", x ) ;
    sum += x ;
}
```

Any of the four sections associated with a for loop may be omitted but the semi-colons must be present always.

For Example :-

```
for ( x = 0; x < 10;    )
    printf( "%d\n", x++ ) ;
...
x = 0 ;
for (    ; x < 10; x++ )
    printf( "%d\n", x ) ;
```

An infinite loop may be created as follows

```
for ( ; ; )
    statement body ;
```

or indeed by having a faulty terminating condition.

Sometimes a for statement may not even have a body to execute as in the following example where we just want to create a time delay.

```
for ( t = 0; t < big_num ; t++ ) ;
```

or we could rewrite the example given above as follows

```
for ( x = 1; x <= 100; printf( "%d\n", x++ ) ) ;
```

The initialisation, condition and increment sections of the for statement can contain any valid C expressions.

```
for ( x = 12 * 4 ; x < 34 / 2 * 47 ; x += 10 )
    printf( "%d ", x ) ;
```

It is possible to build a nested structure of for loops, for example the following creates a large time delay using just integer variables.

```
unsigned int x, y ;

for ( x = 0; x < 65535; x++ )
    for ( y = 0; y < 65535; y++ ) ;
```

For Example :

Program to produce the following table of values

```
#include <stdio.h>

void main()
{
    int j, k ;

    for ( j = 1; j <= 5; j++ )
    {
        for ( k = j ; k < j + 5; k++ )
        {
            printf( "%d  ", k ) ;
        }
        printf( "\n" ) ;
    }
}
```

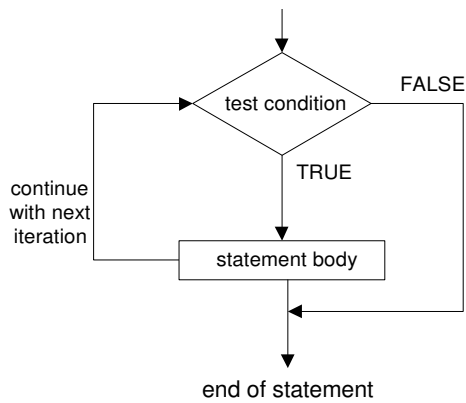
```
1 2 3 4 5
2 3 4 5 6
3 4 5 6 7
4 5 6 7 8
5 6 7 8 9
```

while statement

The ***while*** statement is typically used in situations where it is not known in advance how many iterations are required.

Syntax :

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



For Example : Program to sum all integers from 100 down to 1.

```

#include <stdio.h>
void main()
{
    int sum = 0, i = 100 ;

    while ( i )
        sum += i-- ;// note the use of postfix decrement operator!
    printf( "Sum is %d \n", sum ) ;
}
  
```

where it should be recalled that any non-zero value is deemed TRUE in the condition section of the statement.

do while

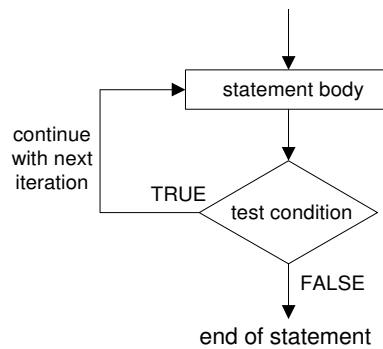
The terminating condition in the for and while loops is always tested before the body of the loop is executed -- so of course the body of the loop may not be executed at all.

In the ***do while*** statement on the other hand the statement body is always executed at least once as the condition is tested at the end of the body of the loop.

Syntax :

```

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



For Example : To read in a number from the keyboard until a value in the range 1 to 10 is entered.

```

int i ;

do
{
    scanf( "%d\n", &i ) ;
} while ( i < 1 && i > 10 ) ;
  
```

In this case we know at least one number is required to be read so the do-while might be the natural choice over a normal while loop.

2.5 break , continue and goto

break statement

When a *break* statement is encountered inside a while, for, do/while or switch statement the statement is immediately terminated and execution resumes at the next statement following the statement.

For Example :-

```

...
for ( x = 1 ; x <= 10 ; x++ )
{
    if ( x > 4 )
        break ;

    printf( "%d " , x ) ;
}
printf( "Next executed\n" ); //Output : "1 2 3 4 Next
Executed"
...
  
```

continue statement

The ***continue*** statement terminates the current iteration of a while, for or do/while statement and resumes execution back at the beginning of the loop body with the next iteration.

For Example :-

```

...
for ( x = 1; x <= 5; x++ )
{
    if ( x == 3 )
        continue ;
    printf( "%d ", x ) ;
}
printf( "Finished Loop\n" ) ; // Output : "1 2 4 5 Finished Loop"
...

```

goto statement

goto statement can be used to branch unconditionally from one point to another in the program. The goto requires a label in order to identify the place where the branch is to be made. A label is any valid variable name and must be followed by a colon (:). The label is placed immediately before the statement where the control is to be transferred.

The syntax is shown below:



Example: Program to detect the entered number as to whether it is even or odd. Use goto statement.

```

#include<stdio.h>
void main()
{
    int x;
    printf("Enter a Number: \n");
    scanf("%d", &x);
    if(x % 2 == 0)
        goto even;
    else
        goto odd;
    even:printf("%d is Even Number");
        return;
    odd:printf(" %d is Odd Number");
}

```

Output:

```

Enter a Number:
5
5 is Odd Number.

```

2.6 Programming examples and exercises.

1. Write a program to find the roots of a user specified quadratic equation.

Recall the roots of $ax^2 + bx + c = 0$ are

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

The user should be informed if the specified quadratic is valid or not and should be informed how many roots it has, if it has equal or approximately equal roots ($b^2 == 4ac$), if the roots are real ($b^2 - 4ac > 0$) or if the roots are imaginary ($b^2 - 4ac < 0$). In the case of imaginary roots the value should be presented in the form ($x + i y$).

2. Write a program that allows the user to read a user specified number of double precision floating point numbers from the keyboard. Your program should calculate the sum and the average of the numbers input. Try and ensure that any erroneous input is refused by your program, e.g. inadvertently entering a non-numeric character etc.

3. Write a program to print out all the Fibonacci numbers using **short** integer variables until the numbers become too large to be stored in a short integer variable i.e. until overflow occurs.

- a. Use a for loop construction.
- b. Use a while loop construction.

Which construction is most suitable ?

Note: Fibonacci numbers are (0, 1,1,2,3,5,8,13,...

4. Write a program which simulates the action of a simple calculator. The program should take as input two integer numbers then a character which is one of $+, -, *, /, \%$. The numbers should be then processed according to the operator input and the result printed out. Your program should correctly intercept any possible erroneous situations such as invalid operations, integer overflow, and division by zero.

MODULE 3

Arrays

3.1 INTRODUCTION

Arrays: Array is a sequential collection of similar data items.

Pictorial representation of an array of 5 integers

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

- An array is a collection of similar data items.
- All the elements of the array share a common name .
- Each element in the array can be accessed by the subscript(or index) and array name.
- The arrays are classified as:
 1. **Single dimensional array**
 2. **Multidimensional array.**

3.1.1 Single Dimensional Array.

- A single dimensional array is a linear list of related data items of same data type.
- In memory, all the data items are stored in contiguous memory locations.

Declaration of one-dimensional array(Single dimensional array)

Syntax:

```
datatype array_name[size];
```

- **datatype** can be int,float,char,double.
- **array_name** is the name of the array and it should be an valid identifier.
- **Size** is the total number of elements in array.

For example:

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

The above statement allocates $5*2=10$ Bytes of memory for the array **a**.

a[0]	a[1]	a[2]	a[3]	a[4]

```
float b[5];
```

The above statement allocates $5*4=20$ Bytes of memory for the array **b**.

- Each element in the array is identified using integer number called as **index**.
- If **n** is the size of array, the array index starts from **0** and ends at **n-1**.

Storing Values in Arrays

- Declaration of arrays only allocates memory space for array. But array elements are not initialized and hence values has to be stored.
- Therefore to store the values in array, there are 3 methods
 1. Initialization
 2. Assigning Values
 3. Input values from keyboard through **scanf()**

3.1.2 Initialization of one-dimensional array

- **Assigning the required values to an array elements before processing is called initialization.**

```
data type array_name[expression]={v1,v2,v3...,vn};
```

Where

- ✓ datatype can be char,int,float,double
 - ✓ array name is the valid identifier
 - ✓ size is the number of elements in array
 - ✓ v1,v2,v3.....vn are values to be assigned.
- Arrays can be initialized at declaration time.

Example:

```
int a[5]={2,4,34,3,4};
```

2	4	34	3	4
---	---	----	---	---

a[0] a[1] a[2] a[3] a[4]

- The various ways of initializing arrays are as follows:
 1. **Initializing all elements of array(Complete array initialization)**
 2. **Partial array initialization**

3. Initialization without size**4. String initialization****1. Initializing all elements of array:**

- Arrays can be initialized at the time of declaration when their initial values are known in advance.
- In this type of array initialization, initialize all the elements of specified memory size.
- Example:

```
int a[5]={10,20,30,40,50};
```

10	20	30	40	50
----	----	----	----	----

2. Partial array initialization

- If the number of values to be initialized is less than the size of array then it is called as partial array initialization.
- In such a case elements are initialized in the order from 0th element.
- The remaining elements will be initialized to **zero automatically by the compiler.**
- Example:

```
int a[5]={10,20};
```

10	20	0	0	0
----	----	---	---	---

3. Initialization without size

- In the declaration the array size will be set to the total number of initial values specified.
- The compiler will set the size based on the number of initial values.
- Example:

```
int a[ ]={10,20,30,40,50};
```

- **In the above example the size of an array is set to 5**

4. String Initialization

- Sequence of characters enclosed within double quotes is called as string.
- The string always ends with NULL character(**\0**)

```
char s[5]="SVIT";
```

We can observe that string length is 4, but size is 5 because to store NULL character we need one more location.

So pictorial representation of an array **s** is as follows:

S	V	I	T	\0
S[0]	S[1]	S[2]	S[3]	S[4]

3.1.2 Assigning values to arrays

Using assignment operators, we can assign values to individual elements of arrays.

For example:

```
int a[3];
    a[0]=10;
    a[1]=20;
    a[2]=30;
```

10	20	30
----	----	----

a[0]

a[1]

a[2]

Reading and writing single dimensional arrays.

To read array elements from keyboard we can use **scanf()** function as follows:

To read **0th** element: `scanf("%d",&a[0]);`

To read **1st** element: `scanf("%d",&a[1]);`

To read **2nd** element: `scanf("%d",&a[2]);`

.....

.....

To read **nth** element : `scanf("%d",&a[n-1]);`

In general

To read **ith** element:

scanf("%d",&a[i]); where **i=0; i<n; i++**

To print array elements we can use **printf()** function as follows:

To print **0th** element: `printf("%d",a[0]);`

To print **1st** element: `printf("%d",a[1]);`

To print **2nd** element :`printf("%d",a[2]);`

.....

.....

To **nth** element : `printf("%d",&a[n-1]);`

In general

To read **ith** element:

printf("%d",a[i]); where **i=0; i<n; i++**

1.	Write a C program to read N elements from keyboard and to print N elements on screen.
	<pre>/* program to read N elements from keyboard and to print N elements on screen */ #include<stdio.h> void main() { int i,n,a[10]; printf("enter number of array elements\n"); scanf("%d",&n); printf("enter array elements\n"); for(i=0; i<n;i++) { scanf("%d",&a[i]); } Printf("array elements are\n"); for(i=0; i<n;i++) { printf("%d",a[i]); } }</pre>
2.	Write a C program to find sum of n array elements .
	<pre>/* program to find the sum of n array elements.*/ #include<stdio.h> void main() { int i,n,a[10],sum=0; printf("enter number of array elements\n"); scanf("%d",&n); printf("enter array elements\n"); for(i=0; i<n; i++) { scanf("%d",&a[i]); } for(i=0; i<n;i++) { sum=sum+ a[i]; }</pre>

	<pre> } printf("sum is %d\n",sum): } </pre>
3.	Write a c program to find largest of n elements stored in an array a.
	<pre> #include<stdio.h> void main() { int i,n,a[10],big; printf("enter number of array elements\n"); scanf("%d",&n); printf("enter array elements\n"); for(i=0; i<n;i++) { scanf("%d",&a[i]); } big=a[0]; for(i=0; i<n;i++) { if(a[i]>big) big=a[i]; } printf("the biggest element in an array is %d\n",big); } </pre>
4.	Write a C program to generate Fibonacci numbers using arrays.
	<pre> #include<stdio.h> void main() { int i,n,a[10]; a[0]=0; a[1]=1; printf("enter n\n"); scanf("%d",&n); if(n==1) { printf("%d\t",a[0]); } if(n==2) { printf("%d\t %d\t",a[0],a[1]); } if(n>2) </pre>

	<pre> { printf("%d \t %d\t",a[0],a[1]); for(i=2;i<n;i++) { a[i]=a[i-1]+a[i-2]; printf("%d\t",a[i]); } } </pre>
--	---

3.2 Searching

- The process of finding a particular item in the large amount of data is called searching.
- The element to be searched is called key element.

There are two methods of searching:

- 1] Linear search.
- 2] Binary search.

1] Linear Search:

- Linear search also called sequential search is a simple searching technique.
- In this technique we search for a given key item in linear order i.e,one after the other from first element to last element.
- The search may be successful or unsuccessful.
- If key item is present, the search is successful, otherwise unsuccessful search.

1.	<p>Program to implement linear search.</p> <pre> #include<stdio.h> void main() { int i,n,a[10],key; clrscr(); printf("enter array elements\n"); scanf("%d",&n); printf("enter array elements\n"); for(i=0; i<n;i++) { scanf("%d",&a[i]); } printf("enter the key element\n"); scanf("%d",&key); for(i=0; i<n;i++) {if(key==a[i]) </pre>
-----------	--

```

        {
            printf("successful search\n");
            exit(0);
        }
    }
    printf("unsuccessful search\n");
}

```

Advantages of linear search

- Very simple Approach.
- Works well for small arrays.
- Used to search when elements are not sorted.

Disadvantages of linear search

- Less efficient if array size is large
- If the elements are already sorted, linear search is not efficient.

2] Binary Search:

- Binary search is a simple and very efficient searching technique which can be applied if the items are arranged in either ascending or descending order.
- In binary search first element is considered as low and last element is considered as high.
- Position of middle element is found by taking first and last element is as follows.

$$\text{mid} = (\text{low} + \text{high}) / 2$$
- Mid element is compared with key element, if they are same, the search is successful.
- Otherwise if key element is less than middle element then searching continues in left part of the array.
- If key element is greater than middle element then searching continues in right part of the array.
- The procedure is repeated till key item is found or key item is not found.

	write a C program to perform binary search on the array of integers
	<pre> /* C program to search a name in a list of names using Binary searching technique*/ #include<stdio.h> void main() { int i, n, low, high, mid,a[50],key; printf("enter the number of elements\n"); scanf("%d",&n); printf("enter the elements\n"); for(i=0;i<n;i++) { Scanf("%d",&a[i]); } printf("enter the key element to be searched\n"); scanf("%d",&key); </pre>

```
low=0;
high=n-1;
while(low<=high)
{
    mid=(low+high)/2;
    if(key==a[mid])
    {
        printf("successful search\n");
        exit(0);
    }
    if(key<a[mid])
    {
        high=mid-1;
    }
    else
    {
        low=mid+1;
    }
}
printf("unsuccessful search\n");
}
```

Advantages of binary search

1. Simple technique
2. Very efficient searching technique

Disadvantages

1. The elements should be sorted.
2. It is necessary to obtain the middle element, which are stored in array. If the elements are stored in linked list, this method cannot be used.

3.3 Sorting

- The process of arranging elements in either ascending order or descending order is called Sorting.

Bubble Sort

- This is the simplest and easiest sorting technique.
- In this technique two successive elements of an array such as $a[j]$ and $a[j+1]$ are compared.
- If $a[j] > a[j+1]$ they are exchanged, this process repeats till all elements of an array are arranged in ascending order.
- After each pass the largest element in the array is sinks at the bottom and the smallest element in the array is bubble towards top. So this sorting technique is also called as sinking sort and bubble sort.

PROGAM FOR BUBBLE SORT

```
#include<stdio.h>
void main()
{
    int a[20],n,temp,i,j;
    printf("enter the number of elements\n");
    scanf("%d",&n);
    printf("enter the unsorted array elements\n");
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);
    for(i=1;i<n;i++)
    {
        for(j=0;j<n-i;j++)
        {
            if( a[ i ] > a[ i+1 ] )
            {
                temp=a[j];
                a[j]=a[j+1];
                a[j+1]=temp;
            }
        }
    }
    printf("the sorted elements arer\n");
    for(i=0;i<n;i++)
        printf(" %d\t", a[i]);
}
```

Write a C program to evaluate the polynomial using Horner's method.


```

#include<stdio.h>
void main()
{
    int i,x,n,a[10],sum;
    printf("enter n:\n");
    scanf("%d",&n);
    printf("enter n+1 co efficient\n");
    for(i=0; i<=n;i++)
    {
        scanf("%d",&a[i]);
    }
    sum=a[n]* x;
    for(i=n-1; i>0; i--)
    {
        sum=(sum+a[i]) *x;
    }
    sum=sum+a[0];
    printf("sum of polynomial equation is %d",sum);
}

```

3.4 Two Dimensional arrays:

- In two dimensional arrays, elements will be arranged in rows and columns.
- To identify two dimensional arrays we will use two indices(say i and j) where I index indicates row number and j index indicates column number.

3.4.1 Declaration of two dimensional array:

```
data_type array_name[exp1][exp2];
```

Or

```
data_type array_name[row_size][column_size];
```

- **data_type** can be int,float,char,double.
- **array_name** is the name of the array.
- **exp1 and exp2** indicates number of rows and columns

For example:

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

- ✓ The above statements allocates memory for $3*4=12$ elements i.e $12*2=24$ bytes.

3.4.2 Initialization of two dimensional array

Assigning or providing the required values to a variable before processing is called initialization.

```

Data_type array_name[exp1][exp2]={
    {a1,a2,...an},
    {b1,b2, .bn},
    .....
    .....
    {z1,z2,...zn}
};

```

- Data type can be int,float etc.
- exp1 and exp2 are enclosed within square brackets .
- both exp1 and exp2 can be integer constants or constant integer expressions(number of rows and number of columns).
- a1 to an are the values assigned to 1st row ,
- b1 to bn are the values assigned to 2nd row and so on.

Example:

```

int a[3][3]={
    {10,20,30},
    {40,50,60},
    {70,80,90}
};

```

10	20	30
40	50	60
70	80	90

Partial Array Initialization

- If the number of values to be initialized is less than the size of array, then the elements are initialized from left to right one after the other.
- The remaining locations initialized to zero automatically.
- Example:

```

int a[3][3]={
    {10,20},
    {40,50},
    {70,80}
};

```

10	20	0
40	50	0

➤ 70	80	0
------	----	---

1.	Write a c program to read & print 2d array as a Array.
	<pre> #include<stdio.h> void main() { int m,n,i,j,a[3][3]; printf("enter number of rows and columns\n"); scanf("%d %d",&m,&n); printf("eneter array elements\n"); for(i=0;i<m;i++) { for(j=0;j<n;j++) { scanf("%d",&a[i][j]); } } printf("array elements are\n"); for(i=0;i<m;i++) { for(j=0;j<n;j++) { printf("%d",a[i][j]); } printf("\n"); } } </pre>
2	Write a c program to add two matrices.
	<pre> #include<stdio.h> void main() { int m,n,i,j,a[3][3],b[3][3] ,c[3][3]; printf("enter number of rows and columns\n"); scanf("%d %d",&m,&n); printf("enter array a elements\n"); for(i=0;i<m;i++) { for(j=0;j<n;j++) { scanf("%d",&a[i][j]); } } } </pre>

```

printf("enter array b elements\n");
for(i=0;i<m;i++)
{
    for(j=0;j<n;j++)
    {
        scanf("%d",&b[i][j]);
    }
}

for(i=0;i<m;i++)
{
    for(j=0;j<n;j++)
    {
        c[i][j]=a[i][j]+b[i][j];
    }
}
printf("resultant matrix c is \n");

for(i=0;i<m;i++)
{
    for(j=0;j<n;j++)
    {
        printf("%d\t",c[i][j]);
    }
    printf("\n");
}
}

```

3 Write a c program to copy one 2d array in to another 2d array

```

#include<stdio.h>
#include<conio.h>
void main()
{
    int m,n,i,j,a[3][3],b[3][3];
    clrscr();
    printf("enter number of rows and columns\n");
    scanf("%d %d",&m,&n);
    printf("enter array a elements\n");
    for(i=0; i<m; i++)
    {
        for(j=0; j<n; j++)
        {
            scanf("%d", &a[i][j]);
        }
    }

    for(i=0;i<m;i++)
    {
        for(j=0;j<n;j++)

```

```

        {
            b[i][j]=a[i][j];
        }
    }
    printf("matrix b is \n");

    for(i=0;i<m;i++)
    {
        for(j=0;j<n;j++)
        {
            printf("%d\t",b[i][j]);
        }
        printf("\n");
    }
}

```

4 Write a c program to find biggest element in a matrix or 2D array.

```

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

    int m,n,i,j,a[3][3];
    clrscr();
    printf("enter number of rows and columns\n");
    scanf("%d %d",&m,&n);
    printf("enter array elements\n");
    for(i=0;i<m;i++)
    {
        for(j=0;j<n;j++)
        {
            scanf("%d",&a[i][j]);
        }
    }
    big=a[0][0];
    for(i=0;i<m;i++)
    {
        for(j=0;j<n;j++)
        {
            if(big>a[i][j])
                big=a[i][j];
        }
    }
    printf("big is %",big);
}

```

5	<p data-bbox="368 212 1136 248">Write a C program to implement Matrix Multiplication</p> <pre data-bbox="368 277 1362 2031">#include<stdio.h> void main() { int m,n,i,j,sum,p,q,k,a[3][3],b[3][3],c[3][3]; printf("enter number of rows and columns of matrix a \n"); scanf("%d %d",&m,&n); printf("enter number of rows and columns of matrix b \n"); scanf("%d %d",&p,&q); if(n!=p) { printf("multiplication not possible\n"); exit(0); } printf("enter matrix a elements\n"); for(i=0;i<m;i++) { for(j=0;j<n;j++) { scanf("%d",&a[i][j]); } } printf("enter array b elements\n"); for(i=0;i<p;i++) { for(j=0;j<q;j++) { scanf("%d",&b[i][j]); } } for(i=0;i<m;i++) { for(j=0;j<q;j++) { { c[i][j]=0; for(k=0;k<n;k++) { c[i][j]= c[i][j]+a[i][k]*b[k][j]; } } } } printf("resultant matrix a is \n"); for(i=0;i<m;i++)</pre>
---	---

```

        {
            for(j=0;j<n;j++)
            {
                printf("%d\t",a[i][j]);
            }
            printf("\n");
        }
        printf("resultant matrix a is \n");

        for(i=0;i<p;i++)
        {
            for(j=0;j<q;j++)
            {
                printf("%d\t",b[i][j]);
            }
            printf("\n");
        }
        printf("resultant matrix a is \n");

        for(i=0;i<m;i++)
        {
            for(j=0;j<q;j++)
            {
                printf("%d\t",c[i][j]);
            }
            printf("\n");
        }
    }
}

```

6 Write a program to find sum of each row and sum of each column

```

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

    int m,n,i,j,rsum,csum,a[3][3];
    printf("enter number of rows and columns\n");
    scanf("%d %d",&m,&n);
    printf("enter array elements\n");
    for(i=0;i<m;i++)
    {
        for(j=0;j<n;j++)
        {
            scanf("%d",&a[i][j]);
        }
    }
    for(i=0;i<m;i++)
    {
        rsum=0;

```

```
        for(j=0;j<n;j++)
        {
            rsum=rsum+a[i][j];
        }
        printf("sum is %d",rsum);
    }

    for(i=0;i<m;i++)
    {
        csum=0;
        for(j=0;j<n;j++)
        {
            csum=csum+a[i][j];
        }
        printf("sum is %d",csum);
    }
}
```

7 Write a C program to add all 2D array elements

```
#include<stdio.h>
void main()
{
    int m,n,i,j,sum=0,a[3][3];
    printf("enter number of rows and columns\n");
    scanf("%d %d",&m,&n);
    printf("enter array elements\n");
    for(i=0;i<m;i++)
    {
        for(j=0;j<n;j++)
        {
            scanf("%d",&a[i][j]);
        }
    }
    for(i=0;i<m;i++)
    {
        sum=sum+a[i][j];
    }
    printf("sum is %d",rsum);
}
```


FUNCTIONS

- A function as series of instructions or group of statements with one specific purpose.
- A function is a program segment that carries out some specific, well defined task.
- A function is a self contained block of code that performs a particular task.

4.1 Types of functions

- C functions can be classified into **two** types,
 1. Library functions /pre defined functions /standard functions /built in functions
 2. User defined functions

1. Library functions /pre defined functions /standard functions/Built in Functions

- These functions are defined in the **library of C compiler** which are used frequently in the C program.
- These functions are written by designers of c compiler.
- C supports many built in functions like
 - Mathematical functions
 - String manipulation functions
 - Input and output functions
 - Memory management functions
 - Error handling functions
- EXAMPLE:
 - `pow(x,y)`-computes x^y
 - `sqrt(x)`-computes square root of x
 - `printf()`- used to print the data on the screen
 - `scanf()`-used to read the data from keyboard.

2. User Defined Functions

- The functions written by the programmer /user to do the specific tasks are called user defined function(UDF's).
- The user can construct their own functions to perform some specific task. This type of functions created by the user is termed as User defined functions.

4.2 Elements of User Defined Function

The Three Elements of User Defined function structure consists of :

1. **Function Definition**
2. **Function Declaration**
3. **Function call**

1. Function Definition:

- A program Module written to achieve a specific task is called as function definition.
- Each function definition consists of two parts:
 - i. **Function header**
 - ii. **Function body**

General syntax of function definition

Function Definition Syntax	Function Definition Example
<pre> datatypefunctionname(parameters) { declaration part; executable part; return statement; } </pre>	<pre> void add() { int sum,a,b; printf("enter a and b\n"); scanf("%d%d",&a,&b); sum=a+b; printf("sum is %d",sum); } </pre>

i. Function header

Syntax

datatype functionname(parameters)

- It consists of **three** parts

a) Datatype:

- ✓ **The data type can be int,float,char,double,void.**
- ✓ This is the data type of the value that the function is expected to return to calling function.

b) functionname:

- ✓ The name of the function.

- ✓ It should be a valid identifier.

c) parameters

- ✓ The parameters are list of variables enclosed within parenthesis.
- ✓ The list of variables should be separated by comma.

Ex: **void add(int a, int b)**

- In the above example the return type of the function is **void**
- the name of the function is **add** and
- The parameters are '**a**' and '**b**' of type integer.

ii. **Function body**

- The function body consists of the set of instructions enclosed between { **and** } .
- The function body consists of following three elements:
 - a) **declaration part:** variables used in function body.
 - b) **executable part:** set of Statements or instructions to do specific activity.
 - c) **return :** It is a keyword, it is used to **return control back to calling function.**

If a function is not returning value then statement is:

return;

If a function is returning value then statement is:

return value;

2. **Function Declaration**

- The process of declaring the function before they are used is called as function declaration or function prototype.
- function declaration Consists of the data type of function, name of the function and parameter list ending with semicolon.

Function Declaration Syntax
datatypefunctionname(type p1,type p2,.....type pn);
Example
int add(int a, int b);
void add(int a, int b);

Note: The function declaration should end with a semicolon ;

3. Function Call:

- The method of calling a function to achieve a specific task is called as function call.
- A function call is defined as **function name followed by semicolon.**
- A function call is nothing but invoking a function at the required place in the program to achieve a specific task.

Ex:

```
void main()  
{  
    add( ); // function call without parameter  
}
```

4.3 Formal Parameters and Actual Parameters

• Formal Parameters:

- The variables defined in the **function header of function definition** are called **formal** parameters.
- All the variables should be separately declared and each declaration must be separated by **commas.**
- The formal parameters **receive the data from actual parameters.**

• Actual Parameters:

- The variables that are used when a function is invoked (in function call) are called **actual** parameters.
- Using actual parameters, the data can be transferred from calling function to the called function.
- The corresponding **formal** parameters in the **function definition** receive them.
- The **actual** parameters and **formal** parameters must match in number and type of data.

- **Differences between Actual and Formal Parameters**

Actual Parameters	Formal Parameters
Actual parameters are also called as argument list . Ex: add(m,n)	Formal parameters are also called as dummy parameters . Ex: int add(int a, int b)
The variables used in function call are called as actual parameters	The variables defined in function header are called formal parameters
Actual parameters are used in calling function when a function is called or invoked Ex: add(m,n) Here, m and n are called actual parameters	Formal parameters are used in the function header of a called function. Example: int add(int a, int b) { } Here, a and b are called formal parameters.
Actual parameters sends data to the formal parameters Example:	Formal parameters receive data from the actual parameters.

4.4 Categories of the functions

1. **Function with no parameters and no return values**
2. **Function with no parameters and return values.**
3. **Function with parameters and no return values**
4. **Function with parameters and return values**

1. **Function with no parameters and no return values**

1. Function with no parameters and no return values (void function without parameter)	
Calling function	Called function
<pre> /*program to find sum of two numbers using function*/ #include<stdio.h> void add(); void main() { add(); } </pre>	<pre> void add () { int sum; printf("enter a and b values\n"); scanf("%d%d",&a,&b); sum=a+b; printf("\n The sum is %d", sum); return; } </pre>
<ul style="list-style-type: none"> ✓ In this category no data is transferred from calling function to called function, hence called function cannot receive any values. ✓ In the above example, no arguments are passed to user defined function add(). ✓ Hence no parameter are defined in function header. ✓ When the control is transferred from calling function to called function a ,and b values are read, they are added, the result is printed on monitor. ✓ When return statement is executed ,control is transferred from called function/add to calling function/main. 	

2. Function with parameters and no return values (void function with parameter)	
Calling function	Called function
<pre> /*program to find sum of two numbers using function*/ #include<stdio.h> void add(int m, int n); void main() { int m,n; printf("enter values for m and n:"); scanf("%d %d",&m,&n); add(m,n); } </pre>	<pre> void add(int a, int b) { int sum; sum = a+b; printf("sum is:%d",sum); return; } </pre>
<ul style="list-style-type: none"> ✓ In this category, there is data transfer from the calling function to the called function using parameters. ✓ But there is no data transfer from called function to the calling function. ✓ The values of actual parameters m and n are copied into formal parameters a and b. ✓ The value of a and b are added and result stored in sum is displayed on the screen in called function itself. 	

3. Function with no parameters and with return values

Calling function	Called function
<pre> /*program to find sum of two numbers using function*/ #include<stdio.h> int add(); void main() { int result; result=add(); printf("sum is:%d",result); } </pre>	<pre> int add() /* function header */ { int a,b,sum; printf("enter values for a and b:"); scanf("%d %d",&a,&b); sum= a+b; return sum; } </pre>

- ✓ In this category **there is no data transfer from the calling function to the called function.**
- ✓ But, there is data transfer from called function to the calling function.
- ✓ No arguments are passed to the function add(). So, no parameters are defined in the function header
- ✓ When the function returns a value, the calling function receives one value from the called function and assigns to variable result.
- ✓ The result value is printed in calling function.

4. Function with parameters and with return values	
Calling function	Called function
<pre> /*program to find sum of two numbers using function*/ #include<stdio.h> int add(); void main() { int result,m,n; printf("enter values for m and n:"); scanf("%d %d",&m,&n); result=add(m,n); printf("sum is:%d",result); } </pre>	<pre> int add(int a, int b) /* function header */ { int sum; sum= a+b; return sum; } </pre>
<ul style="list-style-type: none"> ✓ In this category, there is data transfer between the calling function and called function. ✓ When Actual parameters values are passed, the formal parameters in called function can receive the values from the calling function. ✓ When the add function returns a value, the calling function receives a value from the called function. ✓ The values of actual parameters m and n are copied into formal parameters a and b. ✓ Sum is computed and returned back to calling function which is assigned to variable result. 	

4.5 Passing parameters to functions or Types of argument passing

The different ways of passing parameters to the function are:

- ✓ **Pass by value or Call by value**
- ✓ **Pass by address or Call by address**

1. Call by value:

- In call by value, the values of actual parameters are copied into formal parameters.
- The formal parameters contain only a copy of the actual parameters.
- So, even if the values of the formal parameters changes in the called function, the values of the actual parameters are not changed.
- The concept of call by value can be explained by considering the following program.

Example:

```
#include<stdio.h>
void swap(int a,int b);
void main()
{
    int m,n;
    printf("enter values for a and b:");
    scanf("%d %d",&m,&n);
    printf("the values before swapping are m=%d n=%d \n",m,n);
    swap(m,n);
    printf("the values after swapping are m=%d n=%d \n",m,n);
}

void swap(int a, int b)
{
    int temp;
    temp=a;
    a=b;
    b=temp;
}
```

- Execution starts from function main() and we will read the values for variables m and n, assume we are reading 10 and 20 respectively.
- We will print the values before swapping it will print 10 and 20.
- The function swap() is called with actual parameters m=10 and n=20.
- In the function header of function swap(), the formal parameters a and b receive the values 10 and 20.
- In the function swap(), the values of a and b are exchanged.

- But, the values of actual parameters m and n in function main() have not been exchanged.
- The change is not reflected back to calling function.

2. Call by Address

- In Call by **Address**, when a function is called, the addresses of actual parameters are sent.
- In the called function, the formal parameters should be declared as pointers with the same type as the actual parameters.
- The addresses of actual parameters are copied into formal parameters.
- Using these addresses the values of the actual parameters can be changed.
- This way of changing the actual parameters indirectly using the addresses of actual parameters is known as pass by address.

Example:

```
#include<stdio.h>
void swap(int a,int b);
void main()
{
    int m,n;
    printf("enter values for a and b:");
    scanf("%d %d",&m,&n);
    printf("the values before swapping are m=%d n=%d \n",m,n);
    swap(&m,&n);
    printf("the values after swapping are m=%d n=%d \n",m,n);
}

void swap(int*a, int*b)
{
    int temp;
    temp=*a;
    *a=*b;
    *b=temp;
}
```

NOTE:

Pointer: A pointer is a variable that is used to store the address of another variable.

Syntax: datatype *variablename;

Example: int *p;

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

```
void main()
```

```
{
```

```
    int a, *p;
```

```
    p=&a;
```

```
}
```

In the above program **p** is a **pointer** variable, which is storing the address of variable **a**.

Differences between Call by Value and Call by reference

Call by Value	Call by Address
When a function is called the values of variables are passed	When a function is called the addresses of variables are passed
The type of formal parameters should be same as type of actual parameters	The type of formal parameters should be same as type of actual parameters, but they have to be declared as pointers .
Formal parameters contains the values of actual parameters	Formal parameters contain the addresses of actual parameters.

4.6 Scope and Life time of a variable

Scope of a variable is defined as the region or boundary of the program in which the variable is visible. There are two types

(i) Global Scope

(ii) Local Scope

i. Global Scope:

- The variables that are defined outside a block have global scope.
- That is any variable defined in global area of a program is visible from its definition until the end of the program.
- For Example, the variables declared before all the functions are visible everywhere in the program and they have global scope.

ii. Local Scope

- a. The variables that are defined inside a block have local scope.
- b. They exist only from the point of their declaration until the end of the block.
- c. They are not visible outside the block.

❖ Life Span of a variable

- The life span of a variable is defined as the period during which a variable is active during execution of a program.

For Example

- ❖ The life span of a global variable is the life span of the program.
- ❖ The life span of local variables is the life span of the function, they are created.

❖ Storage Classes

- There are following storage classes which can be used in a C Program:

- i. Global variables
- ii. Local variables
- iii. Static variables
- iv. Register variables

i. Global variables:

- These are the variables which are defined before all functions in global area of the program.
- Memory is allocated only once to these variables and initialized to zero.
- These variables can be accessed by any function and are alive and active throughout the program.
- Memory is deallocated when program execution is over.

ii. Local variables(automatic variables)

- These are the variables which are defined within a function.
- These variables are also called as automatic variables.
- The scope of these variables are limited only to the function in which they are declared and cannot be accessed outside the function.

iii. Static variables

- The variables that are declared using the keyword static are called static variables.

- The static variables can be declared outside the function and inside the function. They have the characteristics of both local and global variables.
- Static can also be defined within a function.

Ex:

```
static int a,b;
```

iv. Register variables

- Any variables declared with the qualifier register is called a register variable.
- This declaration instructs the compiler that the variable under use is to be stored in one of the registers but not in main memory.
- Register access is much faster compared to memory access.

Ex:

```
register int a;
```

Recursion

- Recursion is a method of solving the problem where the solution to a problem depends on solutions to smaller instances of the same problem.
- Recursive function is a function that calls itself during the execution.
- The two types of recursion are

1. Direct Recursion

2. Indirect Recursion

1. Direct recursion

- A recursive function that invokes itself is said to have direct recursion.
- For example factorial function calls itself hence it is called direct recursion.

2. Indirect recursion

- A function which contains call to another function which in turn contains calls another function, and so on.

Design of recursive function

- **Any recursive function has two elements:**

- i. **Base case**
- ii. **General case**

i. **Base case**

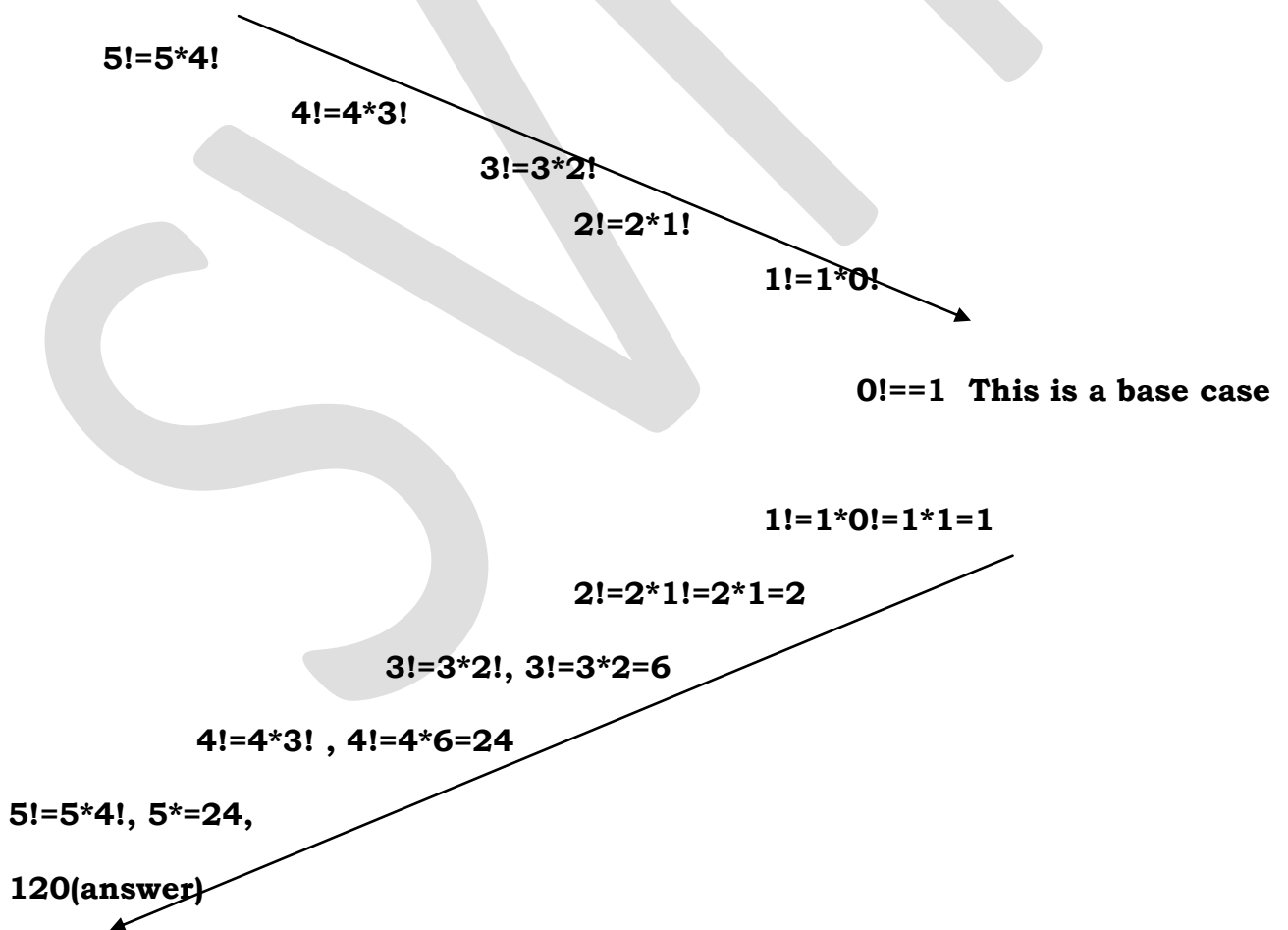
- The statement that solves the problem is called base case.
- Every recursive function must have at least one base case.
- It is a special case whose solution can be obtained without using recursion.

A base case serves two purposes:

- i). It **act as a terminating condition.**
- ii). the recursive function obtains **the solution from the base case it reaches.**

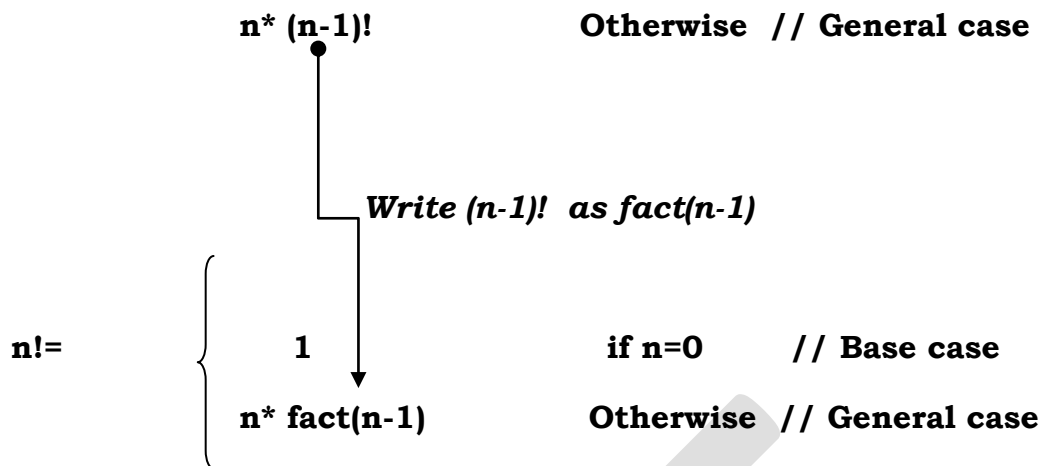
ii. **General case:**

- The statement that reduces the size of the problem is called general case.
- This is done by calling the same function with reduced size.
- In general recursive function of factorial problem can be written as



```

n!=
    {
        1
    }
    if n=0 // Base case
    
```

**Example 1.**

```

/***** Factorial of a given number using Recursion *****/
#include<stdio.h>
int fact(int n);
void main( )
{
    int num,result;
    printf("enter number:");
    scanf("%d",&num);
    result=fact(num);
    printf("The factorial of a number is: %d",result);
}
int fact(int n)
{
    if(n==0)
        return 1;
    else
        return (n*fact(n-1));
}

```


Module 4

Structures And File Management

4.1 Introduction

- Derived data type : int,float,char,double, are the primitive data types .
- Using these primitive data types we can derive other data types and such data types derived from basic types are called **derived data types**.

4.2 Type definition

- Typedef is a keyword that allows the programmer to create new data type name for an already existing datatype.
- Syntax:

```
typedef old datatype newdatatype ;
```

```
Example: typedef int MARKS;  
         typedef float AMOUNT;
```

Example:

- Program to compute simple interest using typedef definition

```
#include<stdio.h>  
typedef float AMOUNT;  
typedef float TIME;  
typedef float RATE;  
void main()  
{  
    AMOUNT p,si;  
    TIME t;  
    RATE r;  
    printf("enter the value of p,t,r\n");  
    scanf("%f%f%f",&p,&t,&r);  
    si=(p*t*r)/100;  
    printf("si=%f\n",si);  
}
```

Advantages of typede:

- 1.Provide a meaningful way of declaring variable
- 2.Increases readability of program
- 3.A complex and lengthy declaration can be reduced to short and meaningful declaration
- 4.Helps the programmer to understand source code easily.

4.3 Structures

- **Definition:** A *structure* is defined as a collection of variables of same data type or dissimilar datatype grouped together under a single name.

Syntax	Example
<pre> struct tagname { datatype member1; datatype member2; datatype member3; } </pre>	<pre> struct student { char name[10]; int usn; float marks; }; </pre>

where

struct is a keyword which informs the compiler that a structure is being defined

tagname: name of the structure

member1,member2: members of structure:

type1,type 2 : int,float,char,double

4.3.1 Structure Declaration

As variables are declared ,structure are also declared before they are used:

Three ways of declaring structure are:

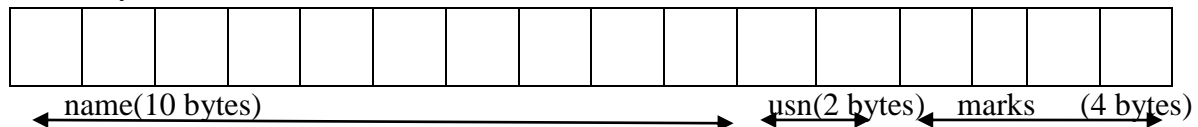
- Tagged structure
- Structure without tag
- Type defined structures

1.Tagged structure	
<pre> syntax struct tag_name { data type member1; data type member2; ----- ----- } ; </pre>	<pre> Example struct student { char name[20]; int usn; float marks; }; </pre>
Declaration of structure variables	
struct tagname v1,v2,v3...vn;	struct student s1,s2,s3;

2.structure without tagname	
syntax struct { data type member1; data type member2; ----- ----- } v1,v2,v3;	Example struct { char name[20]; int usn; float marks; } s1,s2;

3.Type defined structure	
syntax typedef struct { data type member1; data type member2; ----- ----- } TYPE_ID;	Example typedef struct { char name[20]; int usn; float marks; } STUDENT;
Declaring structure variables	
TYPE_ID v1,v2,v3...vn;	STUDENT s1,s2,s3;

Memory Allocation for structure variable s1:



memory allocated for a structure variable s1=memory allotted for name+usn+marks

$$10+2+4$$

16 bytes.

4.3.2 Structure initialization

Syntax:

struct tagname variable={v1,v2....vn};

example

```
struct student s1={"sony",123,24};
```

4.4 Accessing structures

- The members of a structure can be accessed by specifying the variable followed by dot operator followed by the name of the member.
- For example, consider the structure definition and initialization along with memory representation as shown below:

```
struct student
{
    char name[20];
    int usn;
    float marks;

} s1;
struct student s1 = {"aditi",002,40};
```

By specifying**Variblename . membername****Example****S1.name****S1.usn****S1.marks****4.5 Structure operations**

1. Copying of structure variables
2. Arithmetic operations on structures
3. Comparision of two structures

1. Copying of structure variables

- copying of two structure variables is achieved using assignment operator.
- Consider two structure definition of student and employee

<pre>struct student { char name[20]; int usn; float marks; };</pre>	<pre>struct employee { char ename[20]; int eid; float salary; };</pre>
---	--

struct student s1,s2,s3;	struct employee e1,e2,e3;
s1=s2; //valid s2=s1; //valid	
s1=e1;//invalid	

2. Arithmetic operations on structures

- Any arithmetic operation can be performed on the members of a structure Example
- S1.marks++;
- S2.marks++;

3. Comparison of two structures

- variables of same structure definition can be compared along with member.
- S1.usn!=s2.usn;
- S1.marks==s2.marks
- S1.name==s2.name

4.6 Pointers to structure

- A variable which contains address of a structure variable is called pointer to a structure.

<pre>typedef struct { char name[20]; int usn; float marks; } STUDENT; STUDENT s1,s2,s3; STUDENT *p; P=&s1;</pre>	<pre>} Structure definition // structure variables // pointer declaration // assign the address of structure s1 to pointer variable p</pre>
--	---

Access the members of structures using pointers

There are two methods:

1. Dereferencing operator (*)
2. Selection operator (→)

1. Dereferencing operator (*)

- If **p is a pointer** to structure student, members can be accessed as follows:
- (*p).name using this name can be accessed

- (*p).usn using this usn can be accessed
- (*p).marks using this marks can be accessed

2. Selection operator (→)

- If **p is a pointer** to structure student, members can be accessed as follows:
- **p→name** using this name can be accessed
- **p→usn** using this usn can be accessed
- **p→marks** using this marks can be accessed

4.7 Complex structures

1. Nested structures
2. Structure containing arrays
3. Structure containing pointers
4. Array of structures

1. Nested structures

- A structure **which includes another structure** is called nested structure.

<pre> struct subject { int marks1; int marks1; int marks1; }; typedef struct { char name[20]; int usn; <u>struct subject PCD;</u> float avg; } STUDENT; STUDENT s1; </pre>	<pre> } Structure definition with tagname subject //Structure definition with typeid STUDENT // include structure subject with a member name PCD </pre>
---	---

- Structure student has member called PCD, which in turn is a structure .
- Hence **STUDENT** structure is a nested structure.
- We can access various members as follows:
 - S1.name;
 - S1.usn;
 - S1.PCD.marks1;

```
S1.PCD.marks2;
S1.PCD.marks3;
S1.avg;
```

4.8 Array of structure

- As array of integers we can have array of structures
- Suppose we want to store information of 5 students consisting of name,usn,marks,structure definition is as follows:

```
typedef struct
{
    char name[20];
    int usn;
    float marks;
} STUDENT;
STUDENT s[5];
```

If n=3

- We can access the first student information as follows:
 - S[0].name
 - S[0].usn
 - S[0].marks
- We can access the second student information as follows
 - S[1].name
 - S[1].usn
 - S[1].marks
- We can access the second student information as follows
 - S[2].name
 - S[2].usn
 - S[2].marks

hence i ranges from 0 till 2 if n=3

```
for(i=0;i<n;i++)
{
    printf("enter the %d student details\n",i+1);
    printf("enter the roll number:\n");
    scanf("%d",&s[i].rollno);
    printf("enter the student name:\n");
    scanf("%s",s[i].name);
```

```

printf("enter the marks:\n");
scanf("%d",&s[i].marks);
printf("enter the grade:\n");
scanf("%s",s[i].grade);
}

```

Example Programs:

1. Write a c program to store information of n students with name, usn and marks. print the name of the students whose marks is **greater than 90**.

```

#include<stdio.h>
#include<string.h>
#include<stdlib.h>
struct student
{
    char name[20];
    int usn;
    float marks;
};
void main()
{
    int i,n;
    struct student s[10]; // array of structure declaration
    printf("enter the number\n");
    scanf("%d",&n);
    for(i=0; i<n; i++)
    {
        printf("enter the %d student details\n",i+1);
        printf("enter the student name:\n");
        scanf("%s",s[i].name);
        printf("enter the usn:\n");
        scanf("%d",&s[i].usn);
        printf("enter the marks:\n");
        scanf("%d",&s[i].marks);
    }
    for(i=0;i<n;i++)
    {
        if(s[i].marks>90)

```

//check if the marks is greater than 90


```
        { printf("\n marks of the student is %d \n",s[i].marks);
          exit(0);
        }
    }
    printf("student name NOT FOUND\n");
}
```

2. Write a C program to maintain a record of **n** student details using an array of structures with four fields (Roll number, Name, Marks, and Grade). Assume appropriate data type for each field. Print the marks of the student, given the student name as input.

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
struct student
{
    int rollno,marks;
    char name[20];
    char grade[2];
};
void main()
{
    int i,n;
    struct student s[10];
    char key[20];
    clrscr();
    printf("enter the number\n");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        printf("enter the %d student details\n",i+1);
        printf("enter the roll number:\n");
        scanf("%d",&s[i].rollno);
        printf("enter the student name:\n");
        scanf("%s",&s[i].name);
        printf("enter the marks:\n");
        scanf("%d",&s[i].marks);
        printf("enter the grade:\n");
        scanf("%s",&s[i].grade);
    }
    printf("enter the key student name:\n");
    scanf("%s",key);
    for(i=0;i<n;i++)
```

```

    {
        if(strcmp(s[i].name,key)==0)
        {
            printf("\n marks of the student is %d \n",s[i].marks);
            exit(0);
        }
    }
    printf("student name NOT FOUND\n");
}

```

4.9 Structure and functions

➤ The structure or structure members can be passed to functions in various ways as shown below:

Various ways of passing structures to functions:

1. By passing individual members of a structure
2. By passing whole structure
3. By passing structure through pointers

1. By passing individual members of a structure to a function

- A function can be called by passing individual members of a structure as actual parameters.
- Consider the below program which demonstrates the multiplication of fraction:

Program	User defined function to multiply
<pre> #include<stdio.h> typedef struct { int n; int d; }FRACTION; void main() { FRACTION a,b,c; printf("enter fraction 1"); scanf("%d%d", &a.n, &a.d); printf("enter fraction 2"); scanf("%d%d", &b.n, &b.d); /* function call with numerator of a and b*/ c.n=multiply(a.n , b.n); /*same as above function call with denominator of a and b*/ </pre>	<p><i>Continued....</i></p> <pre> int multiply(int x,int y) { return x*y; } </pre>

<pre> c.d=multiply(b.d, b.d); printf("fraction c is %d / %d", c.n, c.d); } </pre> <p style="text-align: center;"><i>Continued to next user defined function....</i></p>	
---	--

2. By passing whole structure

Program	User defined function to multiply
<pre> #include<stdio.h> typedef struct { int n; int d; }FRACTION; void main() { FRACTION a,b,c; printf("enter fraction 1"); scanf("%d %d", &a.n, &a.d); printf("enter fraction 2"); scanf("%d %d", &b.n, &b.d); /* send together numerator and denominator of a and b*/ c=multiply(a , b); //func call printf("fraction c is %d / %d", c.n, c.d); } </pre>	<p><i>Continued....</i></p> <pre> FRACTION multiply(FRACTION x, FRACTION y) { FRACTION z; z.num=x.n*y.n; z.deno=x.d*y.de; return; } </pre>

3. By passing structure through pointers

Program	User defined function to multiply
<pre> #include<stdio.h> typedef struct { int n; int d; }FRACTION; void main() { FRACTION a,b,c; printf("enter fraction 1"); scanf("%d %d", &a.n, &a.d); printf("enter fraction 2"); scanf("%d %d", &b.n, &b.d); } </pre>	<p><i>Continued....</i></p> <pre> void multiply(FRACTION *x, FRACTION *y, FRACTION *z) { z->n = x->n * y->n; z->d = x->d * y->d; return; } </pre>

<pre>/* send together numerator and denominator of a and b*/ multiply(&a , &b ,&c); //func call printf("fraction c is %d / %d", c.n, c.d); }</pre>	
---	--

Uses of structure:

- Structures are used to represent more complex data structure.
- Related data items of dissimilar data types can be grouped under a common name.
- Can be used to pass arguments so as to minimize the number of function arguments.
- When more than one data has to be returned from the function, then structures can be used.
- Extensively used in applications involving database management.

File Handling

4.10:Definitions

File: A file is defined as a collection of data stored on the secondary device such as hard disk. **FILE** is type defined structure and it is defined in a header file “stdio.h”. FILE is a derived data type. FILE is not a basic data type in C language.

Input File: An input file contains the same items that might have typed in from the keyboard.

Output File: An output file contains the same information that might have been sent to the screen as the output from the program.

Text(data) File: A text file is the one where data is stored as a stream of characters that can be processed sequentially.

4.11 Steps to be perform file manipulations

1. Declare a file pointer variable
2. Open a file
3. Read the data from the file or write the data into file
4. Close the file

1. Declare a file pointer variable

- Like all the variables are declared before they are used, a file pointer variable should be declared.
- File pointer is a variable which contains starting address of file.
- It can be declared using following syntax:

FILE *fp;

Example:

```
#include <stdio.h>
void main()
{
    FILE *fp;    /* Here, fp is a pointer to a structure FILE */
    -----
    -----
}
```

2.File open Function

The file should be opened before reading a file or before writing into a file.

Syntax:

```
FILE *fp;
.....
.....
fp = fopen(filename, mode)
```

Where,

fp is a feile pointer
fopen() function to open file
 mode is “**r**,”**w**,”**a**”

- **fopen()** function will return the starting address of opened file and it is stored in file pointer.
- If file is not opened then fopen function returns NULL

```
if (fp == NULL)
{
    printf(“Error in opening the file\n”);
    exit(0);
}
```

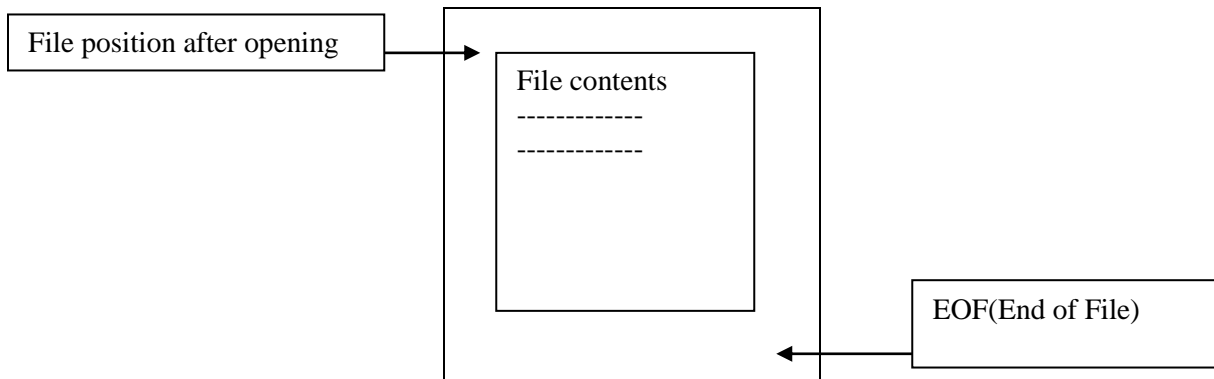
Modes of File

The various mode in which a file can be opened/created are:

Mode	Meaning
“ r ”	opens a text file for reading. The file must exist.
“ w ”	creates an text file for writing.
“ a ”	Append to a text file.

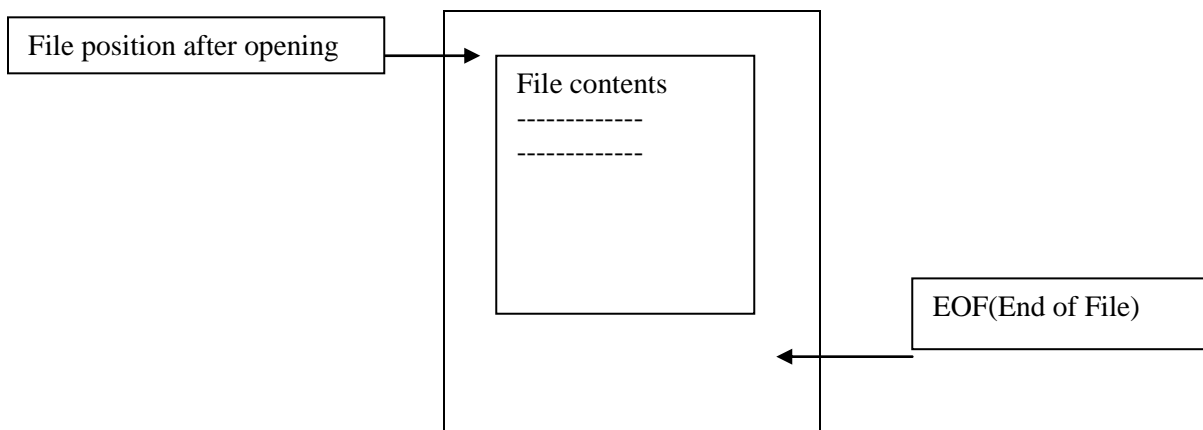
1.Read Mode(“r”)

- This mode is used for opening an existing file to perform read operation.
- The various features of this mode are
 - Used only for text file
 - If the file does not exist, an error is returned
 - The contents of the file are not lost
 - The file pointer points to beginning of the file.



2. Write Mode("w")

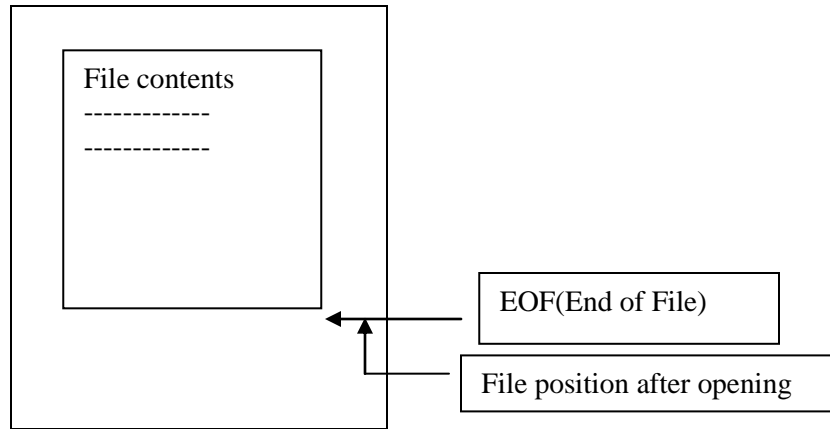
- This mode is used to create a file for writing.
- The various features of this mode are
 - If file does not exist, a new file is created.
 - If a file already exists with the same name, the contents of the file are erased and the file is considered as a new empty file.
 - The file pointer points to the beginning of the file.



3. Append Mode("a")

- This mode is used to insert the data at the end of the existing file.
- The various features of this mode are
 - Used only for text file.
 - If the file already exist, the file pointer points to end of the file.

- If the file is does not exist. A new file is created.
- Existing data cannot be modified.



Examples

Read Mode	Write Mode
<pre>#include<stdio.h> FILE *fp // File Pointer fp=fopen("civil.txt","r"); //opening file civil in read mode if (fp == NULL) //file does not exist { printf("Error in opening the file\n");//error message exit(0); //terminate the program } fclose(fp); // close the file civil.txt</pre>	<pre>#include<stdio.h> FILE *fp // File Pointer fp=fopen("ecb.txt","w"); //opening file ecb in write mode if (fp == NULL) //file does not exist { printf("Error in opening the file\n");//error message exit(0); //terminate the program } fclose(fp); // close the file ecb.txt</pre>

Append Mode
<pre>#include<stdio.h> FILE *fp // File Pointer</pre>


```
fp=fopen("ecb.txt","a"); //opening file ecb in write mode

if (fp == NULL)        //file does not exist
{
printf("Error in opening the file\n");//error message
exit(0);              //terminate the program
}

fclose(fp); // close the file ecb.txt
```

3.Closing a file

- When we no longer need a file, that file should be closed.
- This is the last operation to be performed on a file.
- A file can be closed using **fclose()** function.
- If a file is closed successfully, 0 is returned, otherwise EOF is returned.

Syntax:

fclose(file pointer);

Example:

```
fclose(fp);
```

4.12 I/O(Input and Output) file functions

The three types of I/O functions to read from or write into the file

- I. File I/O functions for **fscanf()** and **fprintf()**
- II. File I/O functions for strings **fgets()** and **fputs()**
- III. File I/O functions for characters **fgetc()** and **fputc()**

File I/O functions for **fscanf()** and **fprintf()**

1.fscanf():

The function **fscanf** is used to get data from the file and store it in memory.

Syntax:

fscanf(fp, "format string", address list);

where,

"fp" is a file pointer.It points to a file from where data is read.

“format String”: The data is read from file and is stored in variable s specified in the list ,will take the values from the specified pointer fp by using the specification provided in format sting.

“address list”:address list of variables

Note:fscanf() returns number of items successfully read by fscanf function.

Example:

<pre>FILE *fp fp=fopen("name.txt","r"); fscanf(fp,"%s",name);</pre>	<pre>FILE *fp fp=fopen("marks.txt","r"); fscanf(fp,"%d%d%d",&m1,&m2,&m3);</pre>
---	---

Note:

- 1.If the data is read from the keyboard then use **stdin** in place of **fp**
- 2.If the data is read from the file then use **fp**

2.fprintf():

The function **fprintf** is used to write data into the file.

Syntax:

fprintf(fp, “format string”, variable list);

where,

“fp” is a file pointer.It points to a file where data to be print.

“format String”: group of format specifiers.

“address list”:list of variables to be written into file

Note:fprintf() returns number of items successfully written by fprintf function.

Example:

<pre>FILE *fp fp=fopen("name.txt","w"); fscanf(fp,"%s",name);</pre>	<pre>FILE *fp fp=fopen("marks.txt","w"); fscanf(fp,"%d%d%d",m1,m2,m3);</pre>
---	--

Note:

- 1.If the data has to be printer on output screen then use **stdout** in place of **fp**
- 2.If the data has to be written to the file then use **fp**

Example Program

Write a C program to read the contents of two files called as name.txt and usn.txt and merge the contents to another file called as output.txt and display the contents on console using fscanf() and fprintf()

```
#include<stdio.h>
#include<stdlib.h>
void main()
{
    FILE *fp1,*fp2,*fp3;
    char name[20];
    int usn;
    fp1=fopen("name.txt","r");
    fp2=fopen("usn.txt","r");
    fp3=fopen("output.txt","w");
    for(;;)
    {
        if(fscanf(fp1,"%s",name)>0)
        {
            if(fscanf(fp2,"%d",&usn)>0)
            {
                fprintf(fp3,"%s %d\n",name,usn);
            }
            else break;
        }
        else break;
    }
    fclose(fp1);
    fclose(fp2);
    fclose(fp3);
    fp3=fopen("output.txt","r");
    printf("NAME\tUSN\n");
    while(fscanf(fp3,"%s %d\n",name, &usn)>0)
    {
        printf("%s \t%d\n",name,usn);
    }
    fclose(fp3);
}
```

File I/O functions for fgets() and fputs()

1.fgets()

fgets() is used to read a string from file and store in memory.

Syntax:

```
ptr=fgets(str,n,fp);
```

where

fp ->file pointer which points to the file to be read

str ->string variable where read string will be stored

n ->number of characters to be read from file

ptr->**If** the operation is successful, it returns a pointer to the string read in.

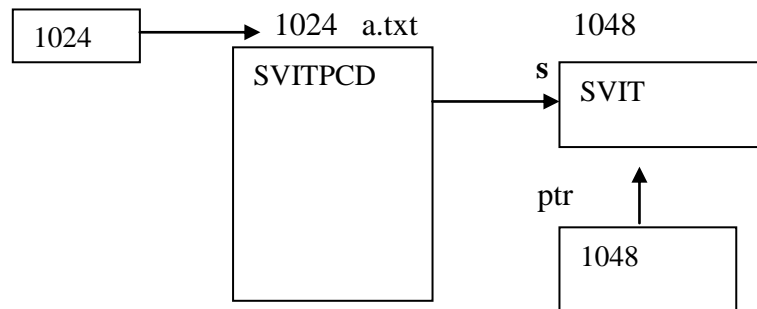
Otherwise it returns NULL.

The returned value is copied into ptr.

Example:

```
FILE *fp;
char s[10];
char *ptr;
fp=fopen("a.txt","r");
```

```
if(fp==NULL)
{
printf("file cannot be opened);
exit(0);
}
ptr=fgets(s,4,fp);
fclose(fp);
```



Example Programs:

<p>1. Write a C program to read from file using function fgets.</p> <pre>#include<stdio.h> void main() { FILE *fp; char str[15]; char *ptr; fp=fopen("name.txt","r"); if(fp==NULL) { printf("file cannot be opened"); exit(0);</pre>	<p>1. Write a C program to read string from keyboard using function fgets.</p> <pre>#include<stdio.h> void main() { char str[15]; char *ptr; printf("Enter the string"); ptr=fgets(str,10,stdin); if(ptr==NULL) { printf("reading is unsuccessful"); exit(0);</pre>
--	---

<pre> } ptr=fgets(str,10,fp); if(ptr==NULL) { printf("reading is unsuccessful"); exit(0); } printf("string is"); puts(str); fclose(fp); } </pre>	<pre> } printf("string is"); puts(str); fclose(fp); } </pre>
--	--

2.fputs()

fputs() is used to write a string into file.

Syntax:

fputs(str,fp);

where

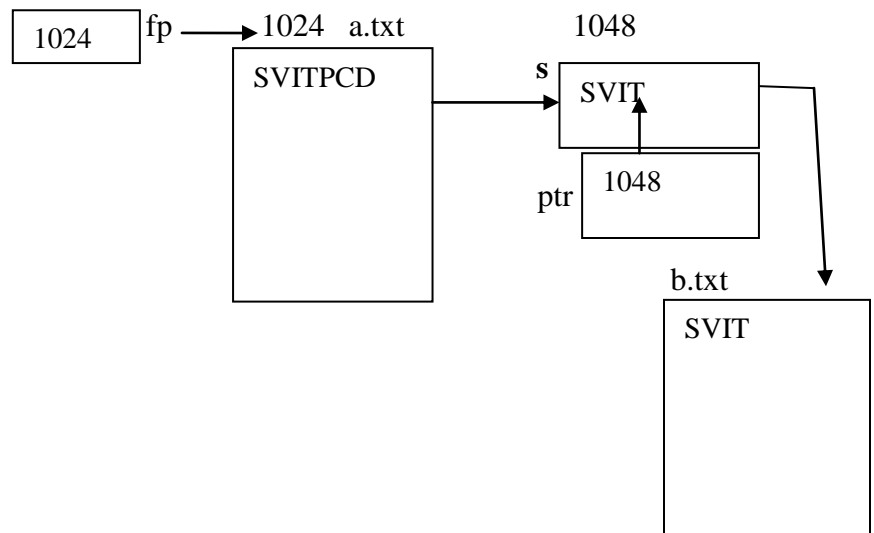
fp ->file pointer which points to the file to be read

str ->string variable where read string will be stored

Example:

```

FILE *fp,*fp1;
char s[10];
char *ptr;
fp=fopen("a.txt","r");
fp1=fopen("b.txt","w");
if(fp==NULL)
{
    printf("file cannot be opened);
    exit(0);
}
ptr=fgets(s,4,fp);
fputs(s,fp1);
fclose(fp);
fclose(fp1);
                
```



Example Program:

1. Write a C program to read from file using function fgets and print into file using fputs function.

```

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

```

FILE *fp,fp1;
char str[15];
char *ptr;
fp=fopen("name.txt","r");
fp1=fopen("output.txt","w");
if(fp==NULL)
{
    printf("file cannot be opened");
    exit(0);
}
ptr=fgets(str,10,fp);
if(ptr==NULL)
{
    printf("reading is unsuccessful");
    exit(0);
}
fputs(str,fp1);
fclose(fp);
fclose(fp1);
}

```

File I/O functions for fgetc() and fputc()

1.fgetc()

fgetc() function is used to read a character from file and store it in memory.

Syntax:

```
ch=fgetc(fp);
```

Example 1:

```

FILE *fp;
fp=fopen("sec.txt","r");
ch=fgetc(fp);

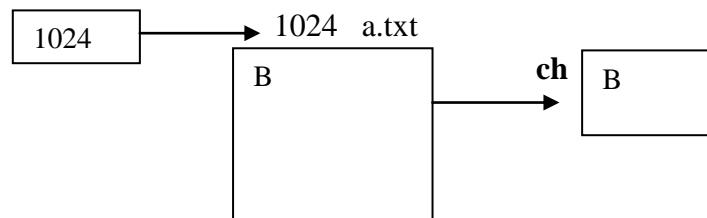
```

Example 2:

```

FILE *fp;
char ch;
fp=fopen("a.txt","r");
ch=fgetc(fp);
fclose(fp);

```



3.fputc()

fgetc() function is used to write a character into a file.

Syntax:

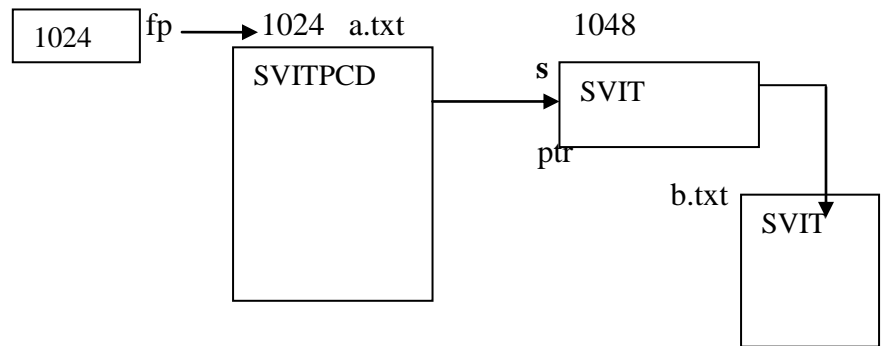
```
fputc(ch,fp);
```

Example 1:

```
FILE *fp;
fp=fopen("sec.txt","w");
fputc(ch,fp);
```

Example 2:

```
FILE *fp,*fp1;
char ch;
fp=fopen("a.txt","r");
fp1=fopen("b.txt","w");
ch=fgetc(fp);
fputc(ch,fp1);
fclose(fp);
fclose(fp1);
```



Example Programs

1. Write a C program to copy one file to another using fgetc() and fputc() functions.

```
#include<stdio.h>
void main()
{
FILE *fp1,*fp2;
char ch;
fp1=fopen("file1.txt","r");
fp2=fopen("file2.txt","w");
while((ch=fgetc(fp1))!=EOF)
{
fputc(ch,fp2);
}
fclose(fp1);
fclose(fp2);
```

2. Write a C program to concatenate two files using fgetc() and fputc() functions.

```
#include<stdio.h>
void main()
{
FILE *fp1,*fp2,*fp3;
char ch;
fp1=fopen("file1.txt","r");
fp2=fopen("file2.txt","r");
fp3=fopen("file3.txt","w");

while((ch=fgetc(fp1))!=EOF)
{
fputc(ch,fp3);
}
```

<pre> }</pre>	<pre> while((ch=fgetc(fp2))!=EOF) { fputc(ch,fp3); } fclose(fp1); fclose(fp2); fclose(fp3); }</pre>
---------------	---

3. Write a C program for counting the characters, blanks, tabs and lines in file.

```

#include<stdio.h>
void main()
{
FILE *fp;
char ch;
int cc=0,bc=0,tc=0,lc=0;
fp1=fopen("file1.txt","r");
while((ch=fgetc(fp1))!=EOF)
{
cc++;
if(ch==' ') bc++;
if(ch=='\n') lc++;
if(ch=='\t') tc++;
}
fclose(fp);
printf("total number of characters=%d\n",cc);
printf("total number of tabs=%d\n",tc);
printf("total number of lines=%d\n",lc);
printf("total number of blanks=%d\n",bc);
}
```

4.13 Command Line Arguments

- The interface which allows the user to interact with the computer by providing instructions in the form of typed commands is called command line interface.
- In the command prompt user types the commands.

Example:

In MS_DOS command prompt looks as follows:

C:\>copy T1.c T2.c

The above copy command copies contents of T1.c to T2.c. In the above line **copy**, T1.c and T2.c are called command line arguments.

Write a C program to accept a file either through command line or as specified by user during runtime and displays the contents.

```
#include<stdio.h>
#include<string.h>
void main(int argc,char *argv[])
{
FILE *fp;
char fname[10];
char ch;
if(argc==1)
{
printf("\n Enter file name\n");
scanf("%s",fname);
}
else
{
strcpy(fname, argv[1]);
}
fp=fopen(fname,"r");
if(fp==NULL)
{
printf("cannot open file");
exit(0);
}
printf("contents of file are\n");
while((ch=fgetc(fp))!=EOF)
{
printf("%c",ch);
}
}
```

Pointers

Introduction

- A pointer is a derived data type. This is the one which stores the address of data in memory, we will be in position to access the data directly and do calculations over it.
- The standard concept is, access the data from memory using variable name it gets the data and operations are done over them.
- But the pointer is different that the accessing is done by address the data is stored so that it will be advantage of decreasing the instructions and overheads of standard usage.

Definition:

A pointer is a variable which contains the address of another variable.

Advantages of pointer

- Enables us to access a variable that is defined outside the function.
- Can be used to pass information back and forth between a function and its reference point.
- More efficient in handling data tables.
- Reduces the length and complexity of a program.
- Sometimes pointers also increases the execution speed.

Declaring of a pointer variable

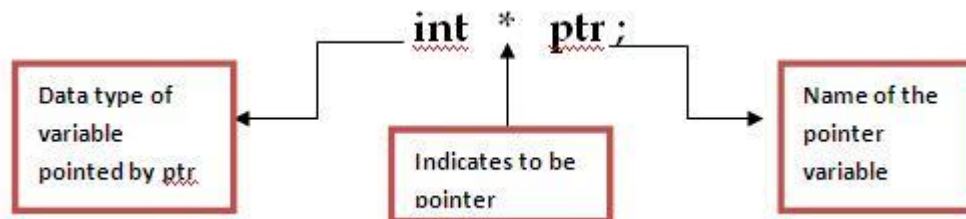
General form:

```
data_type *pointer_name;
```

where,

- The asterisk (*) tells that the variable pointer_name is a pointer variable.
- Pointer_name is a identifier.
- pointer_name needs a memory location.
- pointer_name points to a variable of type data_type which may be int, float, double etc..

Example:



where

- ptr is not an integer variable but ptr can hold the address of the integer variable i.e. it is a **pointer to an integer variable**, but the declaration of pointer variable does not make them point to any location .
- We can also declare the pointer variables of any other data types .

For example:

```
double * dptr;
```

```
char * ch;
```

```
float * fptr;
```

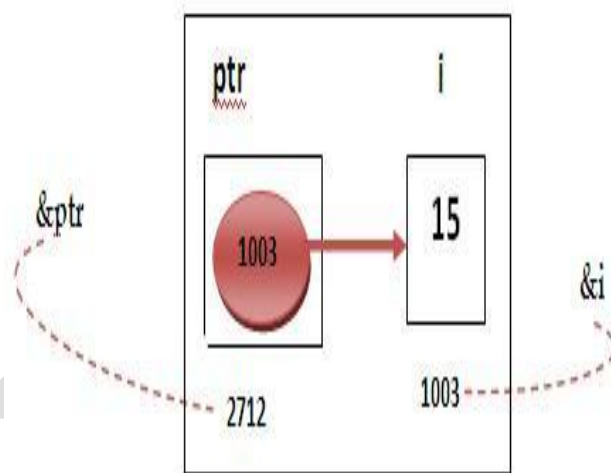
Dereference operator (*)

The unary operator (*) is the dereferencing pointer or indirection pointer, when applied to the pointer can access the value the pointer points to.

Example:

```
int i= 15;
int * ptr;
ptr=&i;
printf("Value of i is :%d",i);
printf("Address of i is :%d",&i);
printf("Value of i is :%d",*ptr);
printf("address of i is :%d",ptr);
printf("address of pointer is :%x",&ptr);
```

```
Value of i is : 15
Address of i is : 1003
Value of i: 15
Address of i is : 1003
Address of ptr is : 2712
```

**The null pointer**

- Sometimes it is useful to make our pointers initialized to nothing. This is called a **null pointer**.
- A null pointer is the pointer does not point to any valid reference or memory address. We assign a pointer a null value by setting it to address 0:

➤ **Example**

```
int *iptr;
iptr=0;    or    iptr = NULL;
```

- This statement assign address 0 to iptr. This statement describe a special preprocessor define called NULL that evaluates to 0.
- Hence A null pointer is a pointer type which has a special value that indicate it is not pointing to any valid reference.

Initialization of pointers

Initializing a pointer variable is a important thing, it is done as follows:

Step 1: Declare a data variable

Step 2: Declare a Pointer variable

Step 3: Assign address of data variable to pointer variable using & operator and assignment operator.

Example:

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

Pointer arithmetic

Pointer arithmetic operations are different from normal arithmetic operations.

The operations allowed to on Pointers are:

1. Add or subtract integers to/from a pointer. The result is a pointer.
2. Subtract two pointers to the same type. The result is an int.
3. Comparison of two pointers

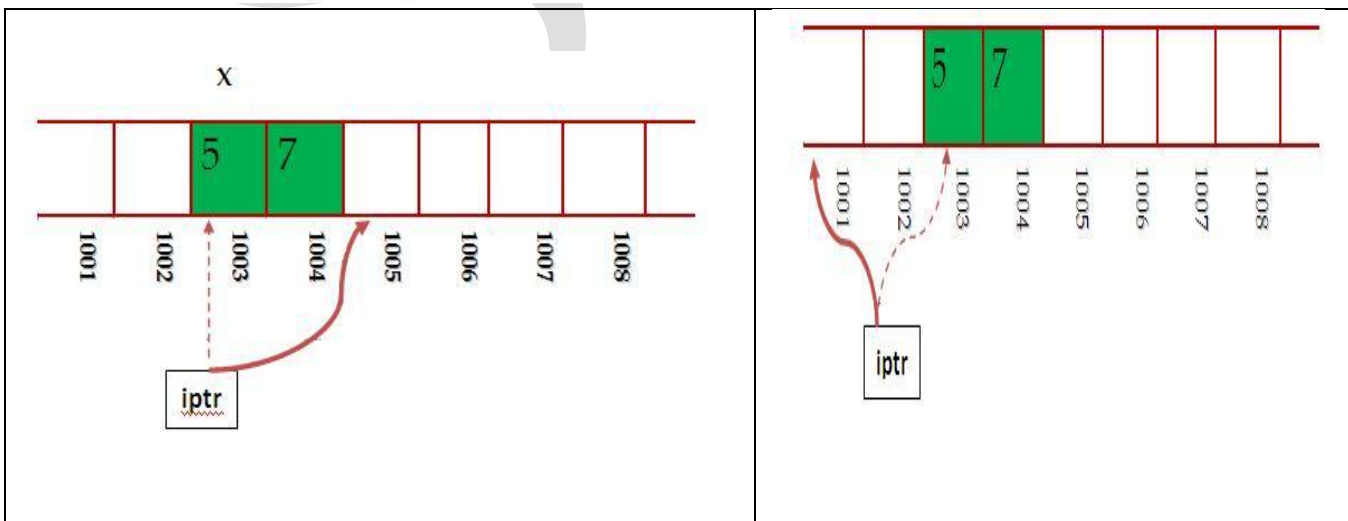
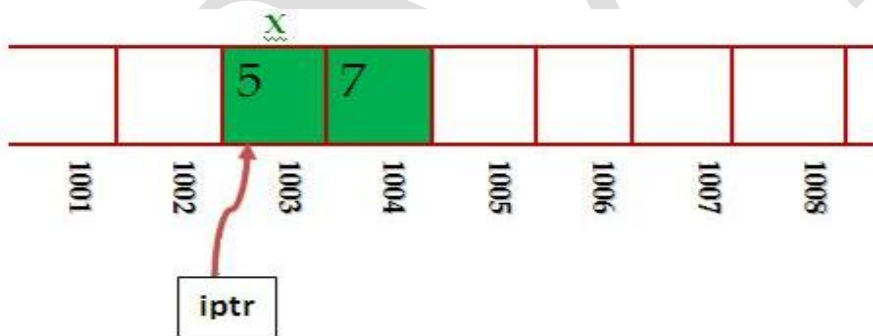
Note: Multiplication, addition, division of two pointers not allowed.

1. Add or subtract integers to/from a pointer

- Simple addition or subtractions operations can be performed on pointer variables.
- If ***iPtr** points to an integer, ***iPtr + 1** is the address of the next integer in memory after ***iPtr**. ***iPtr - 1** is the address of the previous integer before ***iPtr**

Examples:

<pre>int a=57; int *iptr=&a; iptr= iptr +1; iptr = iptr +(2B) iptr =1003+2 iptr =1005</pre>	<pre>int a=57; int *iptr=&a; iptr= iptr -1; iptr = iptr -(2B) iptr =1003-2 iptr =1001</pre>
---	---

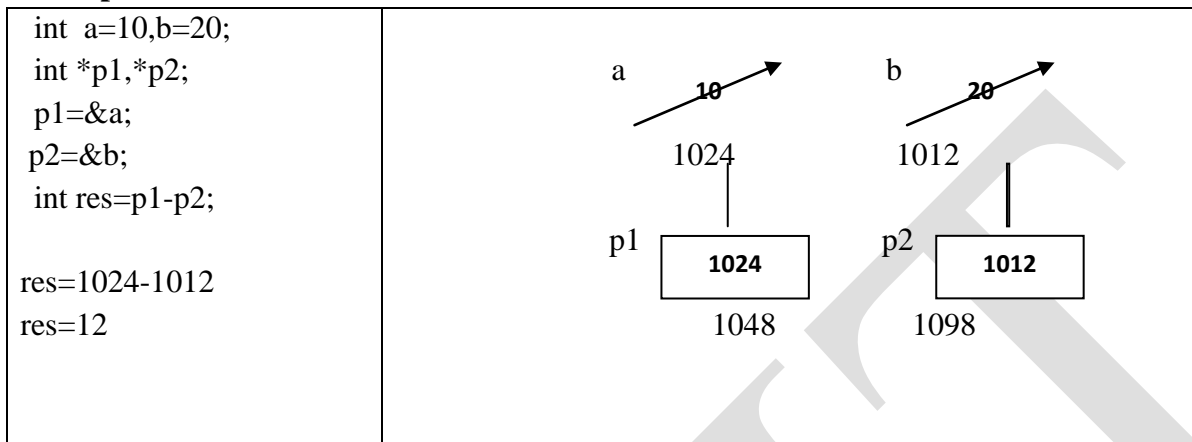


Note: Pointer is always incremented or decremented as per the type of value it is pointing to.

2.Subtract two pointers to the same type.

In C subtraction of two pointers are allowed.

Example:



3.Comparison between two Pointers :

- I. A pointer comparison is valid only if the two pointers are pointing to same array.
- II. All Relational Operators can be used for comparing pointers of same type.
- III. All Equality and Inequality Operators can be used with all Pointer types.
- IV. Two Pointers addition,subtraction and division is not possible.

Pointer Comparison example:

```
#include<stdio.h>
void main()
{
int *ptr1,*ptr2;
ptr1 = (int *)1000;
ptr2 = (int *)2000;
if(ptr2 > ptr1)
printf("Ptr2 is far from ptr1");
}
```

Pointers and functions

- Pointers are often passed to a function as arguments.
- Allows data items within the calling program to be accessed by the function, altered, and then returned to the calling program in altered form.(Called address or by location).
- When arguments are passed to a function by value.
 - The data items are copied to the function.
 - Changes are not reflected in the calling program.

Example:

```
#include<stdio.h>
void swap (int *x, int *y);
void main()
{
    int a, b;
    a = 5;
    b = 20;
    swap (&a, &b);
    printf (“\n a=%d, b=%d”, a, b);
}
void swap (int *x, int *y)
{
    int t;
    t = *x;
    *x = *y;
    *y = t;
}
```

Pointer and arrays

When an array is declared,

- The compiler allocates a base address and sufficient amount of storage to contain all the elements of the array in contiguous memory locations.
- The base address is the location of the first element (index 0) of the array.
- The compiler also defines the array name as a constant pointer to the first element.
- There is a strong relationship between the pointer and the arrays.
- Any operation which can be performed using array subscripting can also be performed using pointers .
- The pointer version will be generally faster than the array version of that program.
- To access the address of any element use (a+i).
- To access the element of array use *(a+i).

Program to read and display array elements using pointers

```
#include<stdio.h>

void main()
{
    int a[100], i, n;
    printf(“enter number of elements”);
    scanf (“%d”, &n);
    printf(“enter array elements”);
    for (i=0; i<n; i++)
```

```

scanf("%d", (a+i));
printf("enter array elements are");
for (i=0; i<n; i++)
    printf("%d", *(a+i));
}

```

Pointers to pointers

- C allows the use of pointers that point to pointers, i.e in turn, point to data (or even to other pointers). In order to do that, add an asterisk (*) for each level of reference in their declarations:

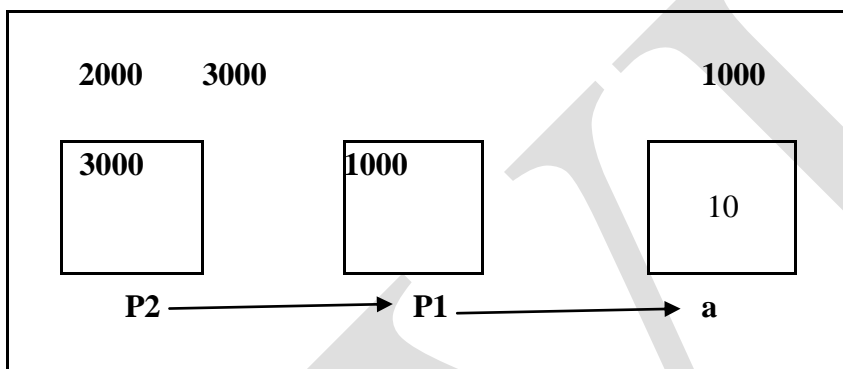
Example

```

int a=10;
int *p1=&a;
int **p2=&p1;

```

Pictorial representation of above example is:



- In the above example p1 is a pointer variable to a and p2 is the pointer to pointer variable for p1.
- p2 has the type int** and a value 3000
- p1 has the type int* and a value 1000
- a has the type int and a value 10

Pointers and structures

- Like array of integers, array of pointer etc, array of structure variables is also available in C.
- And to make the use of array of structure variables efficient, use **pointers of structure type**.

Example:

```

#include <stdio.h>
#include <string.h>
struct student
{
char name[20];
int age;
float percentage;
};
struct student s1={"madan",12,89.5};

```

```
void show_name(struct student *p);
void main()
{
struct student *ptr;
ptr = &s1;
show_name(ptr);
}
void show_name(struct student *p)
{
printf("\n%s ", p->name);
printf("%d ", p->usn);
printf("%f\n", p->percentage);
}
```

Memory Allocation

Memory Allocation in C is of 2 types

1. Static Memory allocation
2. Dynamic Memory allocation

1.Static Memory Allocation

- If the memory is allocated for various variables during compilation time, then it is called as static memory allocation and memory allocated cannot be extended and cannot be reduced, it is fixed.

Example:

```
int a;
int a[10];
int *p;
```

Advantage: efficient execution time.

Disadvantage:

1. The memory is allocated during compilation time. Hence, the memory allocated is fixed and cannot be altered during execution time.
2. Leads to underutilization if more memory is allotted.
3. Leads to overflow if less memory is allocated.

2.Dynamic Memory allocation

- Dynamic memory allocation is the process of allocating memory during run time (execution time).
- Data structures can grow and shrink to fit changing data requirements.
- Additional storage can be allocated whenever needed.
- We can de-allocate the dynamic space whenever we done with them.
- To implementing dynamic memory the following 4 functions are used.

1. malloc(): Allocates requested number of bytes and returns a pointer to the first byte of the allocated space.
2. calloc(): Allocates space for an array of elements, initializes them to zero and then returns a pointer to the memory.
3. realloc(): Modifies the size of previously allocated space.
4. free(): Frees previously allocated space.

1. malloc():

- Allocates requested number of bytes and returns a pointer to the first byte of the allocated space.

Syntax:

```
ptr=(datatype *)malloc(size);
```

where,

- ✓ ptr is a pointer variable of type datatype
- ✓ datatype can be any of the basic datatype or user define datatype
- ✓ Size is number of bytes required.

Example:

```
int *p;  
p=(int *)malloc(sizeof(int));
```

2. calloc():

- It allocates a contiguous block of memory large enough to contain an array of elements of specified size. So it requires two parameters as number of elements to be allocated and for size of each element. It returns pointer to first element of allocated array.

Syntax:

```
ptr=(datatype *)calloc(n,size);
```

where,

- ✓ ptr is a pointer variable of type datatype
- ✓ datatype can be any of the basic datatype or user define datatype
- ✓ n is number of blocks to be allocated
- ✓ Size is number of bytes required.

Example:

```
int *p;  
p=(int *)calloc(sizeof(5,int));
```

3.realloc()

- realloc() changes the size of block by deleting or extending the memory at end of block.
- If memory is not available it gives complete new block.

Syntax:

```
ptr=(datatype *)realloc(ptr,size);
```

where,

- ✓ ptr is a pointer to a block previously allocated memory either using malloc() or calloc()
- ✓ Size is new size of the block.

Example:

```
int *p;  
p=(int *)calloc(sizeof(5,int));  
p=(int *)realloc(p,sizeof(int *8));
```

If enough space doesn't exist in memory of current block to extend, new block is allocated for the full size of reallocation, then copies the existing data to new block and then frees the old block.

4.free()

- This function is used to de-allocate(or free) the allocated block of memory which is allocated by using functions malloc(),calloc(),realloc().

Syntax:

free(ptr);

Note:

It is the responsibility of a programmer to de-allocate memory whenever not required by the program and initialize **ptr** to **Null**.

Difference between Static and Dynamic memory allocation

Static Memory Allocation	Dynamic Memory Allocation
1.Memory is allocated during compilation time	1.Memory is allocated during run time
2.The size of the memory to be allocated is fixed during compilation time and cannot be altered during execution time	2.When required memory can be allocated and when not required memory can be de-allocated
3.Execution is faster	3.Execution is slower
4.Used only when the data size is fixed and known in advance before processing.	4.Can be used when memory requirement is unpredictable.

INTRODUCTION TO DATA STRUCTURES

Data Structure:

- An implementation of abstract data type is data structure i.e. a mathematical or logical model of a particular organization of data is called data structure.
- A data structure deals with the study of how data is organized in memory so that it can be retrieved and manipulated efficiently.

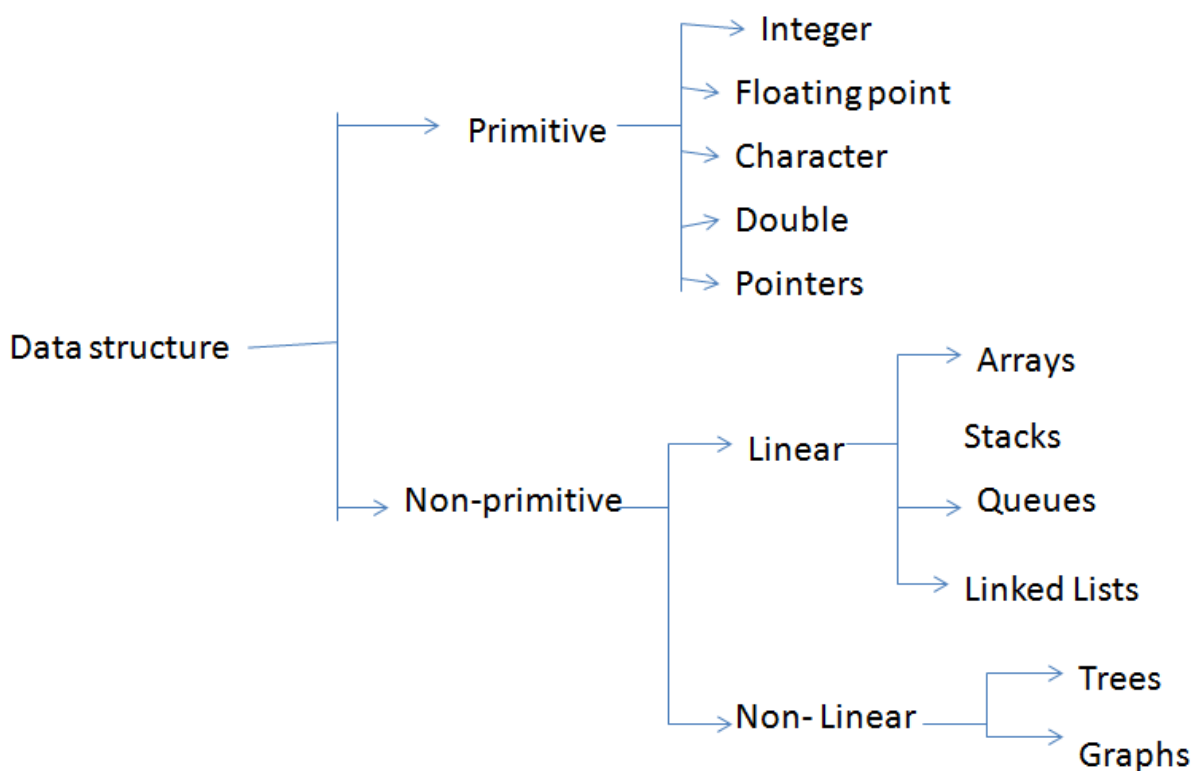
Data structures and its use:

The main focus of data structures are:

- ✓ The study of how data is organized in the memory.
- ✓ How efficiently data can be retrieved and manipulated.
- ✓ The possible ways in which different data items are logically related.

Types of data structure:

A data structure can be broadly classified as shown bellow



I. Primitive data structure

- The data structures, that are directly operated upon by machine level instructions i.e. the fundamental data types such as int, float, double in case of 'c' are known as primitive data structures.

Data Types and Sizes:

There are only a few basic/fundamental data types available in C:

char- a single byte, capable of holding one character in the local character set.

int - an integer, typically reflecting the natural size of integers on the host machine.

float- single-precision floating point.

double- double-precision floating point.

void- does not return any value.

(i) Non-primitive data structure

The data structures, which are not primitive, are called non-primitive data structures.

There are two types of-primitive data structures.

The non-primitive data types cannot be manipulated by machine instructions.

• Linear Data Structures:-

A list, which shows the relationship of adjacency between elements, is said to be linear data structure. The most, simplest linear data structure is a 1-D array, but because of its deficiency, list is frequently used for different kinds of data.

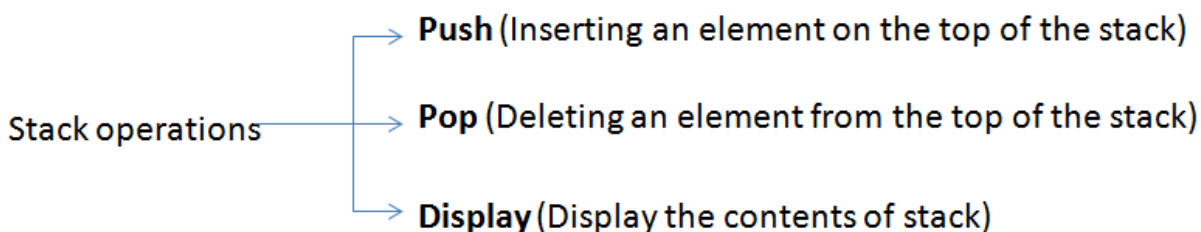
• Non-linear data structure:-

A list, which doesn't show the relationship of adjacency between elements, is said to be non-linear data structure.

Stacks

➔ A stack is a linear data structure in which an element may be inserted or deleted only at one end called the top end of the stack i.e. the elements are removed from a stack in the reverse order of that in which they were inserted into the stack.

➔ A stack follows the principle of last-in-first-out (LIFO) system. The various operations that can be performed on stacks are shown below:



- **Representation of Stacks**

A stack may be represented by means of a one way list or a linear array. We need to define two more variables the size of stack i.e, MAX and the entry or exit of stack i.e., top.

Push: inserting the element to the stack

- * Here we need to check the stack overflow condition which means whether the stack is full.
- * To insert an element two activities has to be done
 1. Increment the top by 1
 2. Insert the element to the stack at the position top.

Pop: deleting an element from the stack

- * Here we need to check the stack underflow condition which means whether the stack is empty or not.
- * To delete the element we need to perform following operations.
 1. Access the top element from the stack.
 2. Decrement top by 1

Display: printing the elements of the stack

- * Here we need to check the stack underflow condition which means whether the stack is empty or not. If empty there will be no elements to display.
- * Display the elements from the 0th position of stack till the stack top.

- **Application of Stacks**

There are two applications of stacks.

- a) Recursion: A recursion function is a function which calls itself. The problems like towers of Hanoi, tree manipulation problems etc can be solved using recursion.
- b) Arithmetic/Evaluation of Expression: The conversion of an expression in the form of either postfix or prefix can be easily evaluated.
- c) Conversions of expressions: Evaluation of infix expressions will be very difficult and hence it needs to be converted to prefix or postfix expression which needs the use of stack.

Queue

- ➔ Queue is a linear data structure in which insertion can take place at only one end called rear end and deletion can take place at other end called top end.
- ➔ The front and rear are two terms used to represent the two ends of the list when it is implemented as queue.

→ Queue is also called First In First Out (FIFO) system since the first element in queue will be the first element out of the queue.

→ Types of queues are

1. Queue (ordinary queue)
2. Circular queue
3. Double ended queue
4. Priority queue

Queue

Here the elements are inserted from one end and deleted from other end. The inserting end is the rear end and the deleting end is the front end.

The operations that can be performed on queue are.

- * Insert an element at rear end
- * Delete an element from the front end
- * Display elements

Circular Queue

In circular queue the elements of queue can be stored efficiently in an array so as to wrap around so that the end of the queue is followed by front of queue.

The operations that can be performed on queue are.

- * Insert an element at rear end
- * Delete an element from the front end
- * Display elements

Double Ended Queue

A Double Ended Queue is in short called as Deque (pronounced as Deck or dequeue). A deque is a linear queue in which insertion and deletion can take place at either ends but not in the middle. The operations that can be performed are

- * Insert an item from the front end
- * Insert an item from rear end.
- * Delete element from front end
- * Delete an element from rear end
- * Display the elements

Priority Queue

A priority queue is a collection of elements such that each element has been assigned a priority value such that the order in which elements are deleted and processed based on the assigned priority.

There are two different types of priority queue

1. Ascending priority queue: Elements can be inserted in any order but deletion is done in the ascending order of values.
2. Descending priority queue: Elements can be inserted in any order but deletion is done in the descending order of values.

Applications of Queues:

- a) Number of print jobs waiting in a queue, when we use network printer. The print jobs are stored in the order in which they arrive. Here the job which is at the front of the queue, gets the services of the network printer.
- b) Call center phone system will use a queue to hold people in line until a service representative is free.
- c) Buffers on MP3 players and portable CD player, iPod playlist are all implemented using the concept of a queue.
- d) Movie ticket counter system, it maintains a queue to take tickets based on first come first serve means who is standing first in a queue, that person will get the ticket first than second person, and so on.

Arrays

Array is a data structure, which is a collection of elements of same datatype.

Advantages :

- * Data accessing is faster.
- * Simple to understand and use

Disadvantages:

- * The size of the array is fixed
- * Array items are stored continuously.
- * Insertion and deletion of an array element in between is a tedious job.

Linked list

A linked list is a data structure which is collection of zero or more nodes with each node consisting of two fields: data and link.

- * Data field consists the information to be processed.
- * Link field contains the address of the next node.

Types of linked list are

1. Singly Linked List
2. Doubly Linked List
3. Circular Singly Linked List
4. Circular Doubly Linked List

Singly Linked List:

A singly linked list is a linked list, where each node has designated field called link field which contains address of next node. Since there is only one link field it is termed as singly linked list.

The various operations of singly linked list are

- * Inserting a node into a list
- * Deleting a node from a list
- * Search in a list
- * Reverse a list
- * Display the content of the list

Circular singly linked list

In the singly linked list the last node link field consists of the NULL(\0), if it contains the address of the first node then it is termed as circular linked list.

Doubly Linked List

It is collection of nodes where each node consists of three fields

- * Info→ which contains the information to be processed
- * Llink→pointer which contains address of the left node or previous node in the list
- * Rlink→ pointer which contains address of the right node link or next node in the list

The main purpose of doubly linked list is its very easy to traverse in any directions either forward or backward direction of the list.

Circularly Doubly linked list

It is the variation of doubly linked list where

- * Rlink of the last node contains address of first node
- * Llink of the first node contains address of the last node

Circularly Doubly linked list with header

It is the variation of the doubly linked list with the header where

- * Llink of header contains address of the last node of the list
- * Rlink of header contains address of the first node of the list
- * Llink of the last node contains address of last but one node of the list
- * Rlink of the last node contains address of the first node of list

Application of Linked List:

- 1) **Polynomial Manipulation like polynomial addition, subtraction, multiplication can be done easily.**
- 2) **Linked Dictionary**
- 3) **Addition of long numbers can be done easily**

Trees

A tree is a nonlinear data structure and is generally defined as a nonempty finite set of elements, called nodes such that:

1. It contains a distinguished node called *root* of the tree.
2. The remaining elements of tree form an ordered collection of zero or more disjoint subsets called subtree

Binary Tree:

A binary tree is defined as a finite set of elements, called nodes, such that:

- 1) Tree is empty (called the null tree or empty tree) or
- 2) Tree contains a distinguished node called root node, and the remaining nodes form an ordered pair of disjoint binary trees

Now let us see **“What are the different ways of traversing the tree?”**

The tree can be traversed in 3 different ways:

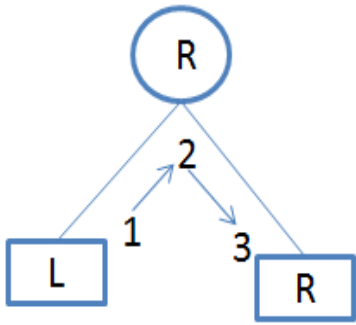
- Inorder traversal
- Preorder traversal
- Postorder traversal

Inorder traversal: Let us see “What is inorder traversing of a binary tree?”

Definition: The inorder traversal of a binary tree can be recursively defined as follows:

1. Traverse the Left subtree in inorder [L]
2. Process the root Node [N]
3. Traverse the Right subtree in inorder [R]

Pictorial representation of inorder traversal is shown below:

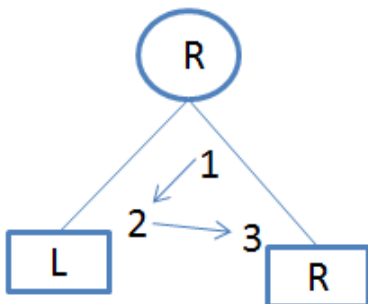


Preorder traversal: Let us see “What is preorder traversing of a binary tree?”

Definition: The preorder traversal of a binary tree can be recursively defined as follows:

1. Process the root Node [N]
2. Traverse the Left subtree in preorder [L]
3. Traverse the Right subtree in preorder [R]

Pictorial representation of preorder traversal is shown below:

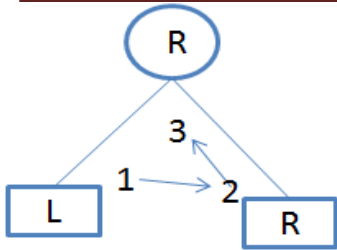


Postorder traversal: Let us see “What is postorder traversing of a binary tree?”

Definition: The postorder traversal of a binary tree can be recursively defined as follows:

1. Traverse the Left subtree in postorder [L]
2. Traverse the Right subtree in preorder [R]
3. Process the root Node [N]

Pictorial representation of postoder traversalis shown below:



Types of binary trees

- * **Complete Binary Tree**

A binary tree is said to be complete if all its level except possibly the last, have maximum number of possible nodes, and if all the nodes at the last level appear as far left as possible.

- * **Full binary tree**

A binary tree said to be full if all its level have maximum number of possible node.

- * **Extended Binary Tree (Strictly Binary Tree or 2-tree)**

A binary tree is said to be extended binary tree if each node has either 0 or 2 children. In this case the leaf nodes are called external nodes and the node with two children are called internal nodes.

- * **Binary Search Trees**

A tree is called binary search tree if each node of the tree has following properties.

The value at a node is greater than every value in the left subtree and is less than every value in the right subtree.

- * **Expression tree**

Application of Binary Tree:

- 1). **Symbol Table Construction:**
- 2). **Manipulation of the Arithmetic Expression**
- 3). **Searching and sorting**
- 4). **syntax analysis of compiler design and to display structure of sentence in a language.**

Introduction to Preprocessor Directives

5.1 Preprocessor Directives:

- Preprocessor Directives are the lines included in the C program that starts with character #
- These are the instructions(also called as directives) to the preprocessor.
- The # symbol is followed by **directive name or an instruction name**.

Advantages of preprocessor are:

- ✓ Programming becomes simple.
- ✓ Program becomes easy to modify and easy to read.

5.2 Types of preprocessor

1. Symbolic names
2. Macros
3. File inclusion
4. Conditional compilation

5.2.1 Symbolic names/symbolic constants

- These are the names which are used to define/give the names for a constant values.
- The # *define* directive is used to define the names for a constant value.
- Since it is used to define constant, it is also termed as **defined constant**.

define name value

Where

- # define is a directive
- name is symbolic name

Example # define MAX 30
 # define PI 3.14

5.2.2 Macros

- Macro is a **name** given to the group of statements.
- When a macro is included in the program, the program replaces the set of instructions defined .
- The # *define* directive is used to **define a macro**.
-

Syntax:

define <macro_name> set of statements

Example1: To find maximum of two variables.

define max (a, b) ((a>b)?a: b)

↓ ↓

macro name (statements for this macro)

Example2 :

```
/*Program to find a square of a number using macro */
```

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

```
#define square (x) ((x)*(x))
```

```
void main( )
```

```
{
```

```
    int m=5;
```

```
    printf("square of m is %d",square(m));
```

```
}
```

In the above example square(x) is the name of the macro which contains x*x as a statement.

5.2.3 File inclusion

- It is a include preprocessor directives
- #include specifies to insert the content of the specified files to the program.
- This directive includes a file in to the code.
- It has two possible forms

```
#include <file>
```

Or

```
#include"file"
```

Example:

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

```
void main()
```

```
{
```

```
    printf("ECE,EEE,CIV\n");
```

```
}
```

5.2.4 Conditional compilation

- The #if, #else ,#endif ,#undef, #ifdef, #ifndef are some of the conditional compilation directives.

#if	it tests a compile time condition
#ifdef	tests for a macro definition
#ifndef	tests whether a macro has been defined or not
#undef	undefines a macro

Example

<pre>#include<stdio.h> void main() { #if((10%2)==0) printf("num is even\n"); #else printf("num is odd\n"); #endif }</pre>	<pre>// num is even is executed</pre>
--	---------------------------------------

SVIT